日本ソフトウェア科学会
第5回プログラミングおよびプログラミング言語
サマースクール

2007-09-12

奈良先端科学技術大学院大学
10:30 – 12:30

非同期リアルタイム処理

～デヴィッド・カトラーの見果てぬ夢～

工学院大学　情報学部　コンピュータ科学科
お　の　　さとし
小 野　　諭
(s-ono@cc.kogakuin.ac.jp)

デヴィッド・カトラーって誰？

- David Neil Cutler (1942～)
- 世界的に著名なOSのアーキテクト、エンジニア。
- 世の中に多大な影響を与えている以下のようなOSを設計・開発した。

RSX-11M: Resource Sharing eXecutive

- DEC 社の 16bit ミニコンピュータ PDP-11 用OS。
- 多重プログラム、メモリ管理をもつリアルタイムOS。
- 1971年より開発を開始し、1974年にリリース

VMS: Virtual Memory System, 後に OpenVMS

- DEC社の 32bit ミニコンピュータ VAX 用OS.
- 仮想記憶（デマンドページング）機能をもつ。
- 1975年より開発を開始し、1978年にリリース。
- 1984年クラスタ、1988年にSMPをサポート。

Windows NT: 後継は、Windows 2000, XP, VIsta..

- マイクロソフト社の 32bit 仮想記憶 OS。(現在は、64bit化)
- 当初は、NT OS/2 として開発を開始したが、Windows NTと改名。
- 1988年に彼がマイクロソフトに転職して、開発を開始。
- V3.1 が最初の版で 1993.7にリリース。日本語版は、V3.5(1994)から。
- 開発の主要なゴールは、下記のとおり。
  - 複数のプラットフォーム(X86, MIPS, Alpha)へのポータビリティ
  - マルチユーザ環境で高度なセキュリティ
  - POSIX への適合性
  - 過去の 16bit Windows ソフトへの後方互換性
  - マルチプロセッサのサポートを含むスケーラビリティ
  - 拡張性
  - 国際化 (Internationalization) の容易性

Cutler’s Kernel

- Cutler の設計したOS カーネルは、RSX-11M から Windows NT 3.1 ~ 4.0 さらに、2000, XP, Vista に至るまで、統一した基本アーキテクチャをもっている。
- もともと、RSX-11Mがいわゆるソフトリアルタイム OSとしての起源をもつこともあり、このカーネルは、以下のような特徴をもつ。
  - カーネルモードへのリアルタイム性の高いユーザ機能の組み込みが容易。
  - プロトコルスタックの組み込みを始め、再起動なしの動的なカーネルモジュールの接続関係変更が可能。
  - カーネルから、ユーザモードまで、非同期処理をベースにする統一したアーキテクチャ。

今回の講演の仮説とねらい

- きれいな Unix、きたない Windows というのは、偏見ではないか。
- WindowsNT のカーネルの基本モデルは、1970年代はじめに David Cutler がデザインしたころから、ほとんど変化していない。
- Thread, 非同期入出力、リアルタイムシグナルなどの増改築を重ねてシンプルさを喪失した　Posix API と比べて、非同期処理主体で、単純さを失わない Windows NT アーキテクチャには、見るべきものが多いと考える。
- 大規模・高速通信、SMP活用などの高性能アプリを作成するさいに、1970年代に起源を発する古くて新しい非同期処理モデルが有効ではないか、と考えている。
- Cutler のアーキテクチャへの入門的・体系的な解説が少なく、過去の知見の伝承が、必ずしも円滑に行われていない。
- 開発対象の要件に合わせて、適切なアーキテクチャを選択できる技術者を育てたい。

ところで、講師はどんな人？

現職: 工学院大学　情報学部　コンピュータ科学科　教授

専門:　分散システム、リアルタイム処理、情報セキュリティなど

残念ながら、David Cutler の知り合いではない。

ソフトウェアやネットワークサービスの研究開発が、楽しくて明るい仕事であることを願い続けてきた。（しかし、現実は、記憶する限り、あまりそうではなかったかもしれない。。。。）

組み込みシステムは、今後の発展が非常に有望な分野であるし、日本の競争力が生かされる分野であると確信している。

今回の講演で、もし、Posix とは一味違うリアルタイム処理構成法の有望性と面白さにめざめて、興味をもって取り組んで下さる方がひとりでも増えたとしたら、大変に光栄である。

- 1977年から、DEC社の16bit ミニコンピュータ PDP-11/10 とリアルタイムOS (RT-11, RSX-11M)に触れてこの世界に入り、魅了される。
- その後、8bitマイクロプロセッサへのCutler モデルに基づく手作りRTOSによる産業用機器の開発経験を経て、VAX/VMS, SPARC/Solaris, Windows NT などのSMPサポート、カーネルマルチスレッド機能のある多機能OS を用いた組み込みシステムの研究開発を業務とする。
- 主に高度通信サービスの研究開発を目的として、ハードウェアから、ファーム、ドライバ、カーネルモジュール、デーモン、アプリという多階層構造のハードウェア・ソフトウェア同時設計システムの開発に従事。
- さらに、Linux の組み込み用途利用に興味をもち、商用サービスのための通信機器開発にも応用。
- 近頃は、ビザンチン障害を考慮した、信頼性や長期的な安全性の高いセキュアなサービスインフラの研究に興味をもっている。

組み込みシステムの主な開発経験:

（開発担当ないし技術責任者のものに限る）

- ユーザスレッドライブラリ (PDP-11/RT-11)
- コンピュータ間通信システム (PDP-11/RSX-11M)
- Intel 8080/8085 向けリアルタイムOS（下記の機器で利用)
- 産業用コンピュータ向けBSC手順通信ボード (自作RTOS / 8085)
- 電力会社変電所開閉器集中制御システム (自作RTOS / 8085)
- リスト処理並列コンピュータ制御システム (VAX/VMS)
- ISDN 広域時計同期装置 (Sun SPARC/SunOS, Solaris)
- マルチアングル広域映像同期システム (Intel / Solaris, MC68360)
- フロー優先度制御 IPSec スタック(カーネルMT) (Intel 4Way/ Solaris)
- VLAN対応多重アドレス空間分離 IPスタック (Intel / FreeBSD)
- PDC/PHS用VPN通信PCカード (独自OS / NEC V830)
- 公衆無線LAN用ユーザ認証・暗号化・トンネリングドライバソフト
  (Intel / Windows 98SE, ME, 2k, XP)
- 公衆無線LAN用ユーザ認証・暗号化・トンネリングPCカード
  (NEC VR4131 (MIPS系) / Linux 2.4)

講師略歴

学歴:

- 1977.3 東京大学工学部 電子工学科 卒業
- 1979.3 東京大学大学院工学系研究科 電気工学専門課程 修士課程 修了
- 1982.3 東京大学大学院工学系研究科 電気工学専門課程 博士課程 修了
- 1982.3 工学博士 (東京大学)

職歴:

- 1982.4 日本電信電話公社 武蔵野電気通信研究所
- 1985.4 日本電信電話(株) 基礎研究所
- 1987.9 日本電信電話(株) ソフトウェア研究所
- 1999.2 日本電信電話(株) 情報流通プラットフォーム研究所
- 2005.4 工学院大学　技術者能力開発センター (CPDセンター)　教授
- 2006.4 工学院大学　情報学部　コンピュータ科学科　教授

現在にいたる。 この間、

- 1991.7 から1年間　東大工学部知能工学寄付講座　客員助教授
- 2002.4 から3年間　東大大学院情報理工学系研究科　客員教授

本日の講演の構成

I. Posix PThread モデルの分析
II. Cutler のモデルの動機と概要

III. SMP とそれを支えるOS
IV. 現在のOS階層の基本モデル
V. David Cutler の処理モデル
VI. デザインパターンの実例
VII. まとめ
付1：Windows NT ファミリーのコンテキスト
付2: Priority Inversion

I. Posix PThread モデルの分析

- 現在、価格低下が進むマルチコア、マルチプロセッサを活用した、高性能アプリ、リアルタイムアプリの開発が強く求められている。
- これまで、こうした基盤として、国際標準化が進んだ Posix Thread (PThread) (IEEE Std 1003.1, 2004)が知られている。
- PThread は、標準化されてはいるものの、この方法で、信頼性・拡張性がある高性能なソフトを開発するのは困難であると思われる。
- 特に、マルチスレッドソフトは、再現性・網羅性のある試験が著しく困難で、数年後にバグが出現する、などということも、しばしば発生する。
- マルチスレッドは、シングルスレッドの正常な進化とは言い切れない。

マルチスレッド化で失う重要な性質 (Lee)

1. 理解性:
2. 予測可能性:
3. 決定性

- マルチスレッドは、まず(そのままでは正しい結果を与えない)広範な非決定性を導入。
- プログラマは、その非決定性の枝狩りを行う。

本来あるべき方法:

- まず、合成可能で決定性のある部品を作成すべき。
- 非決定性は、その上で、明示的・思慮深く導入すべきである。

* Lee, E.A.: The Problem with Treads, IEEE Computer, May 2006 (Vol. 39, No. 5) pp. 33-42

マルチスレッド化に伴う困難

[プログラムの正しい意味の理解の困難]

- 操作のアトミック性喪失に伴う想定外の動作
- 操作の可視性喪失に伴う想定外の動作
- コンパイラのセマンティクスのあいまい性

[プログラムの設計や性能の予測に関する困難]

4. ロックの粒度と性能へのインパクト
5. 非決定性

- スレッドのインタリーブ実行に伴う可能性を正確に追いかけたり、正しく枝狩りすること、それを試験することは難しい。

単一スレッドのソフトで成り立っている性質

```
x = 0;
x = x + 1;
x = x + 1;
```

実行後、x の値は2。
（変数操作のアトミック性)

```
x = 1;
y = x;
```

(store値の可視性)

- 人間は、通常、プログラムを先頭から逐次的に実行しながら、状態の変化を追う。実際の実行がこれと対応していることが重要。
- 単一スレッドの場合、人間の直感的な計算モデルとの対応関係は失われない。
- 言語仕様のあいまい性により、コンパイラとハードウェアPFの組み合わせにより、マルチスレッド下で、予測外の値が計算されることがある。

1.1 変数操作のアトミック性
2スレッドそれぞれによる、1加算演算

複数スレッドが共有データを更新する場合、正しい排他制御を実施しないと、操作のアトミック性が失われて、予測できない結果を生む。

(1) スレッドAが、先行した場合

Thread A	temp	x	temp	Thread B
{				{
register temp:	-	0	-	register temp:
temp = x;	0	0	-	/*zzz…*/
temp++;	1	0	-	/*zzz…*/
x = temp;	1	1	-	/*zzz…*/
}	1	1	1	temp = x;
	1	1	2	temp++;
	1	2	2	x = temp;
				}

0に2回1を加えるので、結果は2になる。

複数スレッドが共有データを更新する場合、正しい排他制御を実施しないと、操作のアトミシティが失われて、予測できない結果を生む。 **(2) スレッドBが、先行した場合**

Thread A	temp	x	temp	Thread B
{				{
register temp:	-	0	-	register temp:
/*zzz…*/	-	0	0	temp = x;
/*zzz…*/	-	0	1	temp++;
/*zzz…*/	-	1	1	x = temp;
temp = x;	1	1	1	}
temp++;	2	1	1	
x = temp;	2	2	1	
}				

この場合も、同様に結果は2になる。

(3) スレッドAの実行とスレッドBの実行がインタリーブした場合

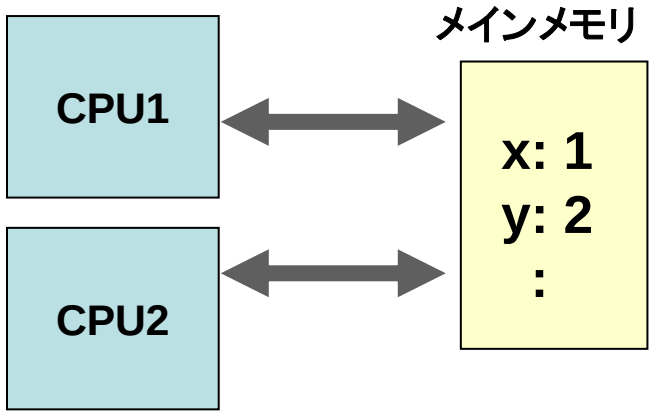
操作のアトミック性が失われると、人間の直感的モデルと異なる結果になる。

Thread A	temp	x	temp	Thread B
{				{
register temp:	-	0	-	register temp:
temp = x;	0	0	-	/*zzz…*/
temp++;	1	0	-	/*zzz…*/
/*zzz…*/	1	0	0	temp = x;
/*zzz…*/	1	0	1	temp++;
/*zzz…*/	1	1	1	x = temp;
x = temp;	1	1	1	/*zzz…*/
}				}

0に1を2回加えたのに、結果は1になってしまう！

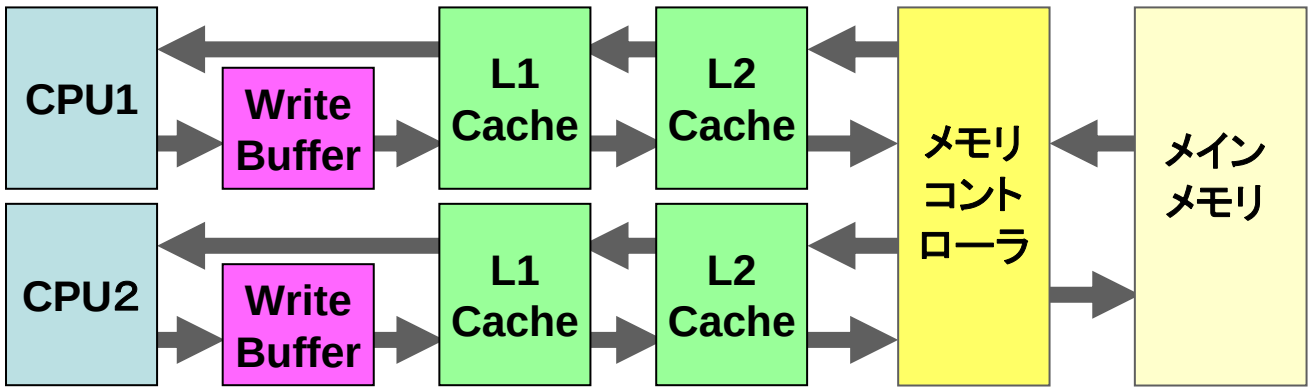
1.2 メモリモデルと操作の可視性

人間のもつ暗黙のCPU/メモリイメージ

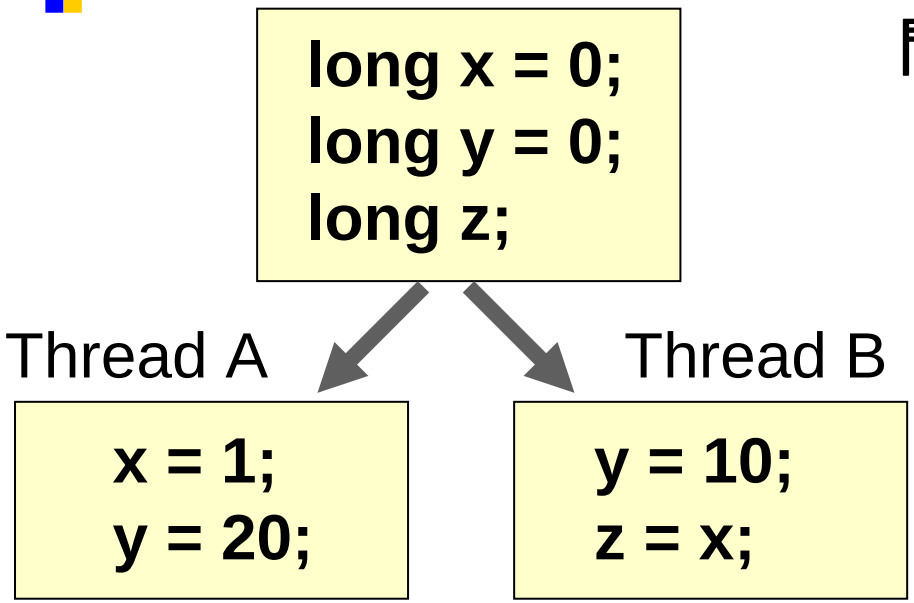


- それぞれのCPUは、独立してメインメモリに読み書き
- メモリは、両者の操作を併合して動作。
- メモリの値は、それぞれの瞬間において、どのCPU からみても、同じ値が見える。

近代的なSMPの典型的なシステムイメージ



•メモリへの書き込みは、Write Buffer や多段のキャッシュで遅延させられる。
•明示的な同期がない場合、操作の可視性はCPUごとに異なり、メモリの値も異なりうる。
•操作の実行順序は、リオーダされうる。



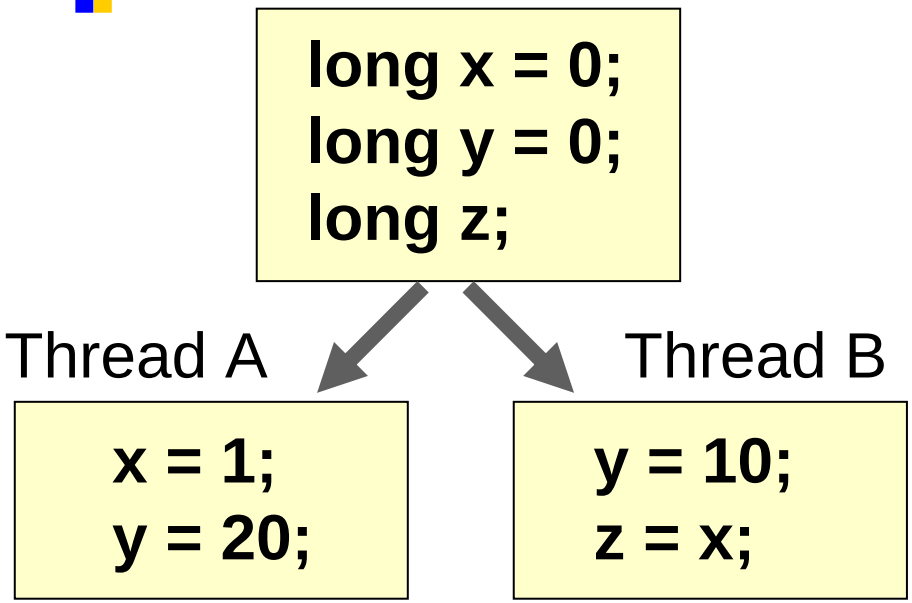
問題:

- 実行の結果、z の値は 0 であった。
- このとき、yの値としてとり得る値は何？
- その理由は？

参考： Multiprocessor Considerations for Kernel-Mode Drivers, Microsoft Windows Hardware and Driver Central, 2004
(ただし、この文献のIA64メモリモデルの記述は、一部正しくないと思われる。)

標準的？な推論方法:

- z の値が0なので、xの値も 0 である。
- これは、スレッドB の2操作が、スレッドA の2操作に先行したことを意味している。
- であるから、yの値は、最終的には後のスレッドAの値が活きて、結果として、y = 20 となる。
- Yの値として、他にとり得る値はない。



問題:

- 実行の結果、z の値は 0 であった。
- このとき、yの値としてとり得る値は何？
- その理由は？

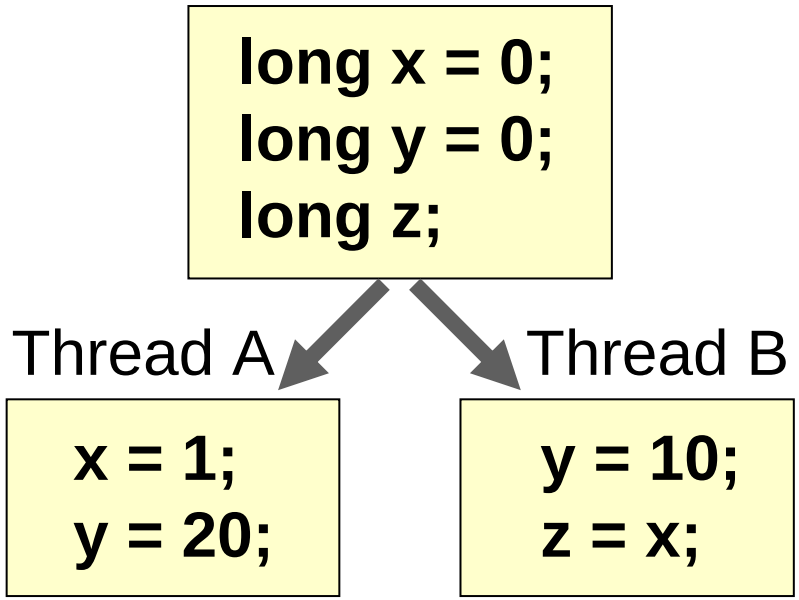
通常の予測？:

- y = 20　のみ。
- 理由:　前ページのとおり。

解答:

- まことに残念かつ驚くべきことに、不正解です。
- 典型的なCPUである x86, x64, Itanium (IA-64) では、y = 20 に加えて、y = 10 もあり得ます。

なぜ、そのようなことが発生するのでしょう？？？？？



- x86, x64, IA64などでは、あるWrite操作と後続する異なる場所へのRead操作がある場合、 Write操作が遅延されて、先にRead 操作が行われることがある。
- x86, x64 では、後続するのがWrite 操作の場合は、順序の入れ替えは発生しない。
- IA64 では、規格上、上記のWrite/Write の場合にも入れ替えが発生しうる。ただし、これまでのIA64系CPU では、入れ替えが実際に観測されたことはない。

A1: Write x
A2: Write y

B2: Read x ← スレッドBで、x として 0 が読み込まれる。

スレッドBで、Write y / Read xで、命令実行順序の入れ替えが発生。

B1: Write y
B3: Write z ← スレッドBで、z として 10 が書き込まれる。

1.3 コンパイラのセマンティクスのあいまい性

- C, C++, Java などの言語のVolatile 型修飾子は、従来、宣言された変数への特定の最適化を禁止するコンパイラへの指示とみなされてきた。
- 具体的には、Volatile 宣言された変数に対して、
  - レジスタへの割り当てを禁止する。
  - Read, Write 操作の省略や投機的な実行を禁止する。
  - Volatile 宣言された変数相互の操作の順序入れ替えを禁止する。
- 一方、Volatile 宣言された変数とされていない変数との操作の順序入れ替えは許容する。
- これらのセマンティクスは、Volatile 変数が、キャッシュされないメモリ空間において、外部レジスタの I/F　として利用されることを想定していたものと推察される。
  （実際、このような用途には、必ず使用されていた。）

- 仕様を誤解して、Volatile 変数を、マルチスレッド下の低レベル（軽量）ロックとして使うことによるバグが頻発したためか、Volatile 宣言のセマンティクスが、近年差し替えられた。
- Java 5.0 および Microsoft Visual Studio 2005 から、Volatile 宣言は、コンパイル時の最適化のみならず、実行時の命令順序入れ替えも規制される。
- すなわち、Volatile 宣言された変数は、ロックで保護された変数に近い動作をする。
- Volatile 宣言に対して、このようなロック的な意味を与えることや、永らく使われてきたセマンティクスを大きく変更することに対する批判もくすぶっている。

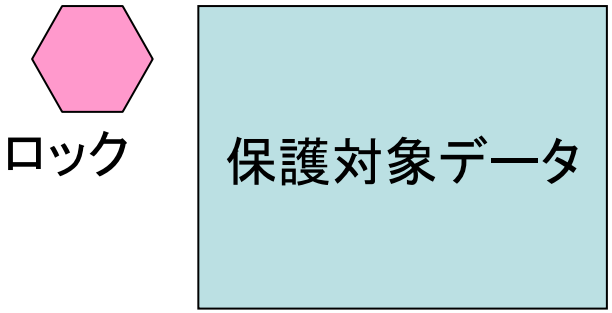
•Brian Goetz: Java theory and practice: Fixing the Java Memory Model, (http://www.ibm.com/developerworks/java/library/j-jtp02244.html)
•Vance Morrison: MEMORY MODELS Understand the Impact of Low-Lock Techniques in Multithreaded Apps (http://msdn.microsoft.com/msdnmag/issues/05/10/MemoryModels/)

マルチスレッドプログラム理解の困難性の原因

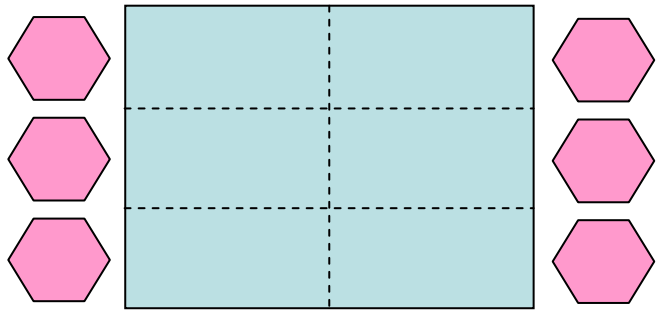
1. スレッドの動作がインタリーブすること。
2. インタリーブの要素となるアトミックな操作が不明確
3. コンパイラによる最適
   - コンパイラは、スレッド単位のセマンティクスで最適化する。
   - スレッド内で不変な値の読み出しの省略
   - スレッド内で不要な値の格納の省略
   - 実行順序の変更
4. ハードウェアによる最適化　(CPUアーキテクチャで異なる)
   - 命令の実行順序の変更
   - キャッシュと主記憶の不一致
5. プログラム言語の意味定義の非厳密性と変更
   - 変数型宣言における Volatile 修飾子のセマンティクス

1.4 ロックの粒度およびスレッド数の問題

粒度の粗いロックの例



粒度の細かいロックの例



要件： コーディングの手間やエラーの発生しやすさ、
実行時オーバヘッド、デッドロック

保護対象データに対して、粗い粒度のロックを用いると、

- 保護対象データの複数個所への操作の一貫性確保が容易。
- ロックの掛け忘れや、掛け違いのエラーが減少する。
- 異なるロックのループによるデッドロックのエラーが減少する。
- ロックの衝突によるオーバヘッドが増大する。

(粒度を細かくすると、利点・欠点が逆転する。)

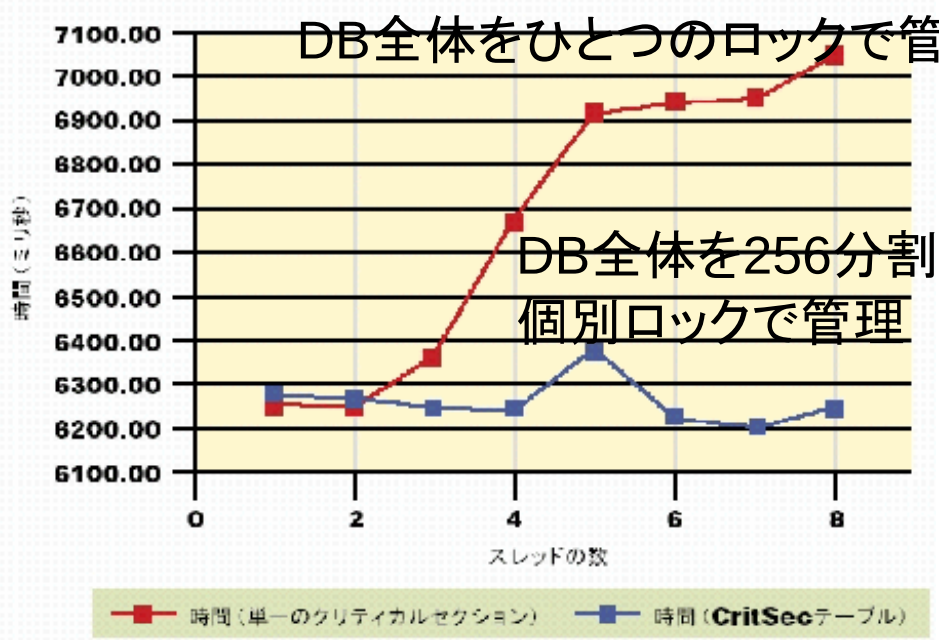
- 同期型処理を用いた場合、応答性を確保するため、スレッド数を増大させる傾向がある。
- 一般に、過大なスレッド生成は、コンテキストSw オーバヘッドとロック衝突から、性能低下を招きやすい。

性能評価例＊:
•キーと文字列値のペアからなるメモリ内DBを模擬。
•DB全体をひとつのクリティカルセクションにする場合と256分割して個別にロックする場合とを比較。
•CPU数を1と4の場合で、処理スレッド数を変化

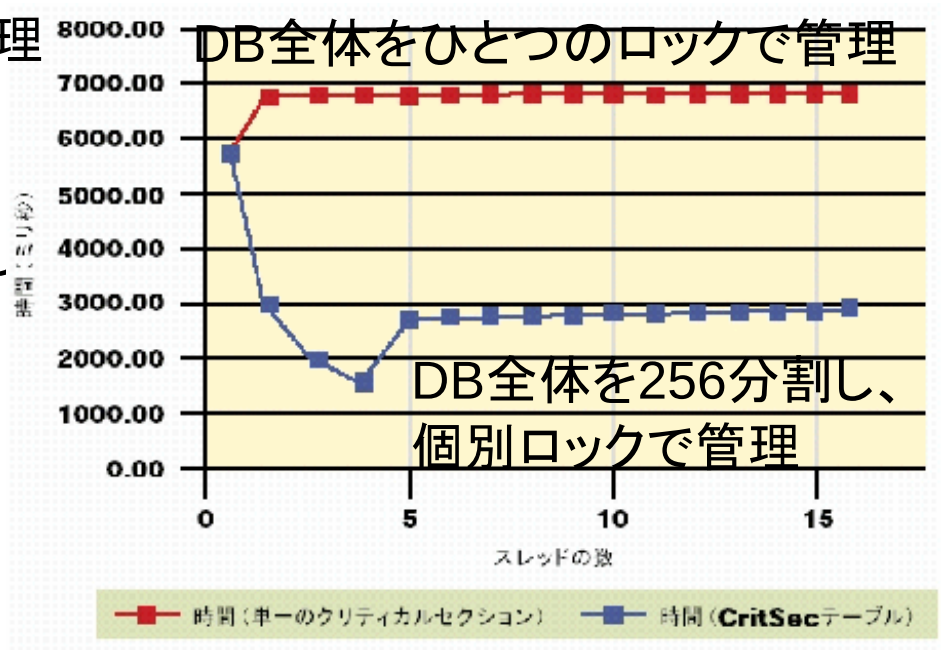
＊Ian Emmons: Multiprocessor Optimizations: Fine-Tuning Concurrent Access to Large Data Collections, MSDN Magazine, August, 2001
(http://msdn.microsoft.com/msdnmag/issues/01/08/Concur/)
(日本語訳 http://www.ascii.co.jp/pb/msdn/article/a18_0021.html)

ロックの粒度およびスレッド数の問題

シングルプロセッサの場合



4プロセッサの場合

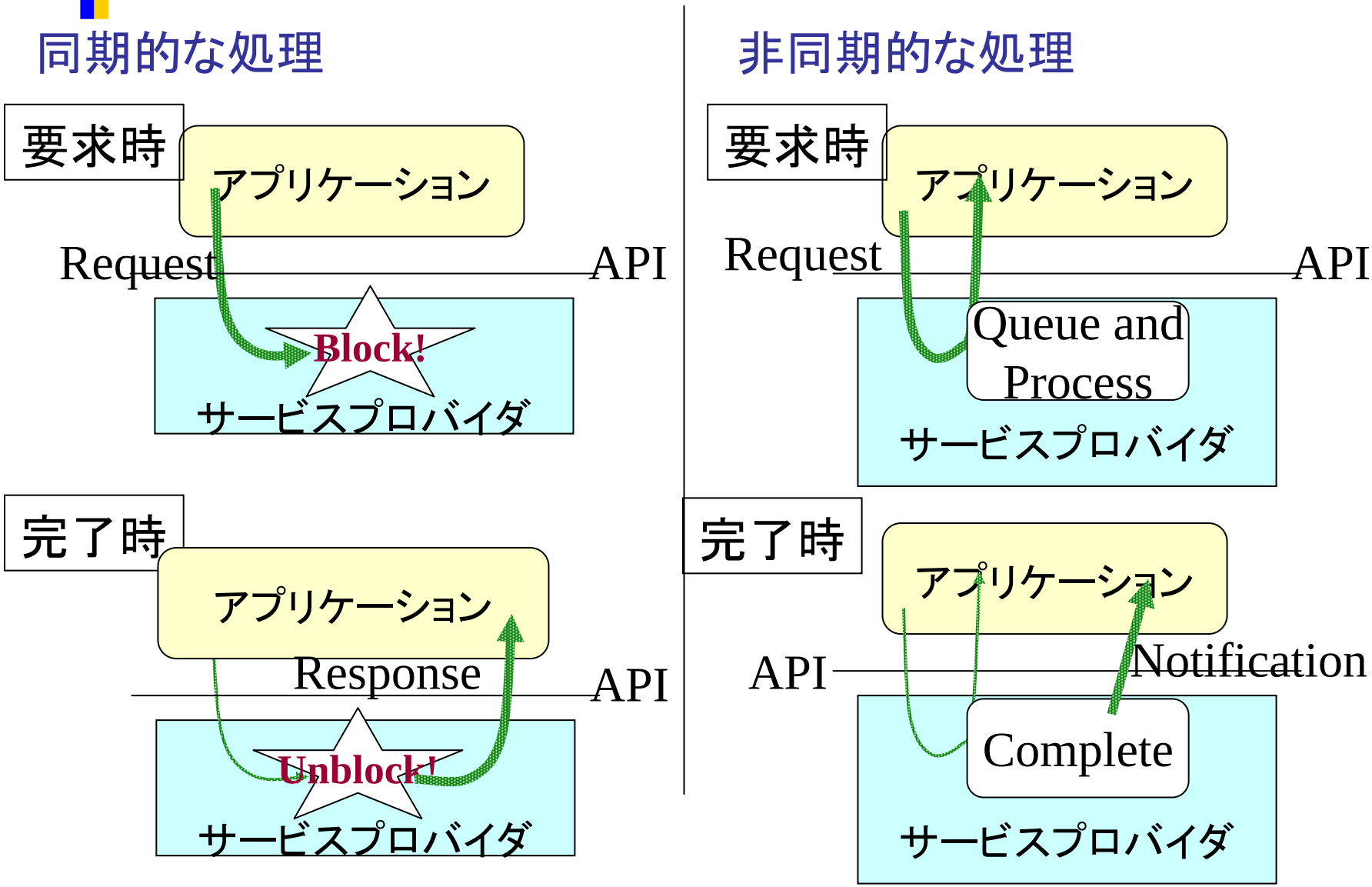


- スレッドを過剰に増やすと、性能が低下する場合がある。
- ロックの粒度が粗すぎると、マルチCPUの速度向上効果が薄い。

1.5. 非決定性

- 多数のスレッドに対する自由度の高いスケジュールの結果、同一データに対する処理経路が常に変化し、試験や不具合の再現を困難にする。
- 同期型処理では、システムの状態は、スレッドのブロッキング位置などで暗黙に表現される。したがって、システム全体の正確な状態が把握しにくい。
- 要求のキャンセルでは、ブロック中、実行中のスレッドの両方に対する正確な対応が必要。（これは非常に困難）
- 同期型処理の不足を補うため、マルチスレッド、スレッド向けシグナル、ノンブロッキングI／O、リアルタイムシグナルなどを併用すると、拡張機能の相互作用により、理解や試験がさらに困難になりやすい。

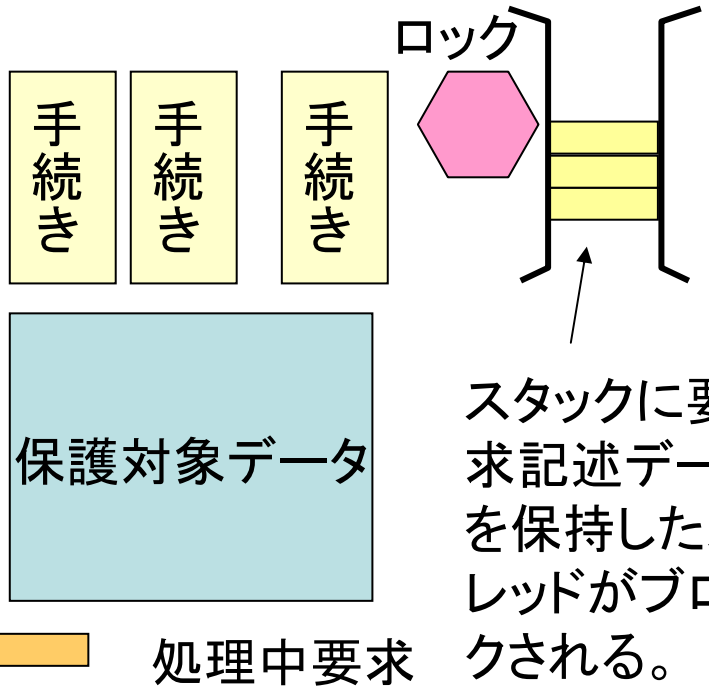
II. Cutler のモデルの動機と概要
非同期処理とそのデザインパターン



同期処理と非同期処理の内部機構の差

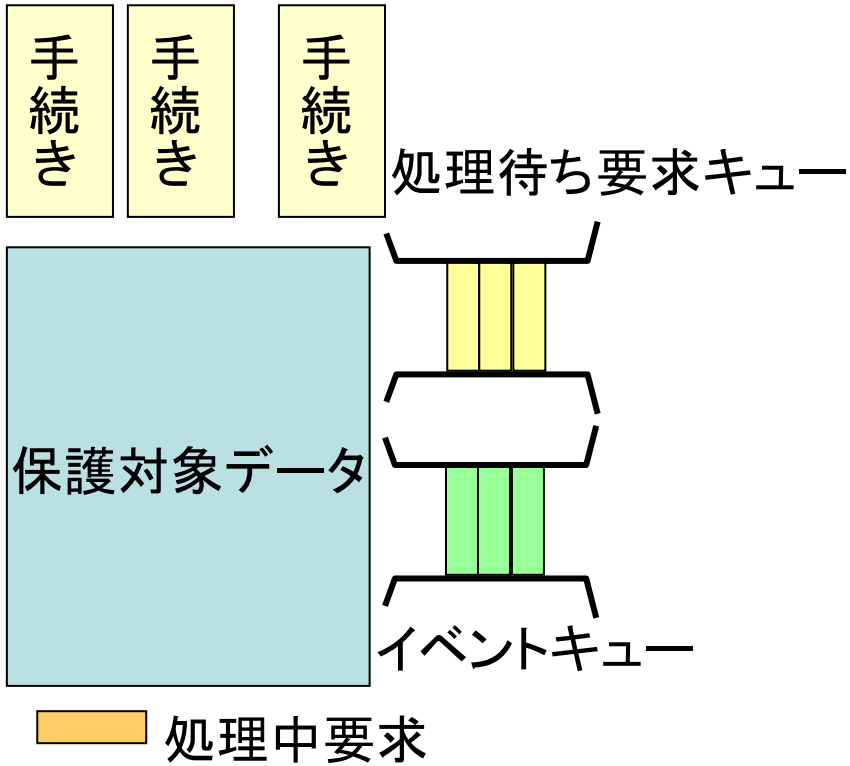
同期的な処理

Synchronized(逐次化）された
オブジェクトアクセスの場合、
ロックにより、スレッドがブロック



非同期的な処理

スレッドではなく、スレッドが出した
要求のみがキューイング、処理される。



Cutler　の方法はどう違うのか？

[PThread 方式]

- 処理主体：複数の独立にスケジュールされる一様な特性のスレッド
  - 実際のCPU数とは無関係な論理的な主体。実CPUに多重化。
- 対象：任意のスレッドから並行して呼び出されうる、リエントラントな手続き。データ保護のために、ロックなどによる逐次化を行う。
- 入出力、外部との通信：　同期型（終了までスレッドをブロック）が基本

[Cutler 方式]

- 処理主体：高度に逐次化された、多様なコンテキストのスレッド
  - 同一手続きへの呼び出し多重度が、実CPU数で制限される。
- 対象：　適合したコンテキストのスレッドのみから逐次呼び出しされる再利用可能な手続き。
  - 呼び出し段階で逐次化が行われるので、付加的なロックは、複数コンテキスト間で共有されるキューヘッダなどに限られる。
  - マルチプロセッサ環境での並行呼び出しの保護に SpinLock を利用。
- 入出力、外部との通信：　非同期型（ノンブロック)で、終了は、イベントまたは、コールバックにて通知

プログラミングスタイル：（後で詳細に説明）

- イベント、コールバックを主体にして状態遷移を記述する方法。
- カーネルやユーザアプリのエントリを、「コンテキスト」で色分けする。
- あるエントリを直接呼び出せるのは、適合したコンテキストで実行しているものに限られる。呼び出し先から再度コールバックされることも可能。

  適合しないエントリは、対応したProxy を経由して呼び出す必要がある。

  （すなわち、どのスレッドからもどのエントリを呼び出せる自由度を捨てている。）
- カーネルにおけるスケジュールポリシーは、かなりの部分がソフトウェアアーキテクチャに組み込まれている。したがって、純粋なスレッド優先度制御のものに比べて、優先度逆転が発生しやすい。

Cutler 型処理モデルの評価

[プログラムの正しい意味の理解の困難]

1. 操作のアトミック性喪失に伴う想定外の動作
   - 逐次呼び出しを基本にしている。
   - 複数スレッドが競合するのは、要求を受け取るキューヘッダなど、ごく一部のデータ構造に限られる。これらには、アトミシティが確保された、専用のインターロック命令が用意されている。
2. 操作の可視性喪失に伴う想定外の動作
   - スレッドのブロックは、非同期処理では発生しない。
   - スレッドのプリエンプティブな切り替えは、コンテキストの変化を伴い、その変化過程において、バリア同期がなされる。
3. コンパイラのセマンティクスのあいまい性
   - 競合発生箇所が非常に限られるので、入出力など本来の目的以外で Volatile のセマンティクスを必要としない。

[プログラムの設計や性能の予測に関する困難]

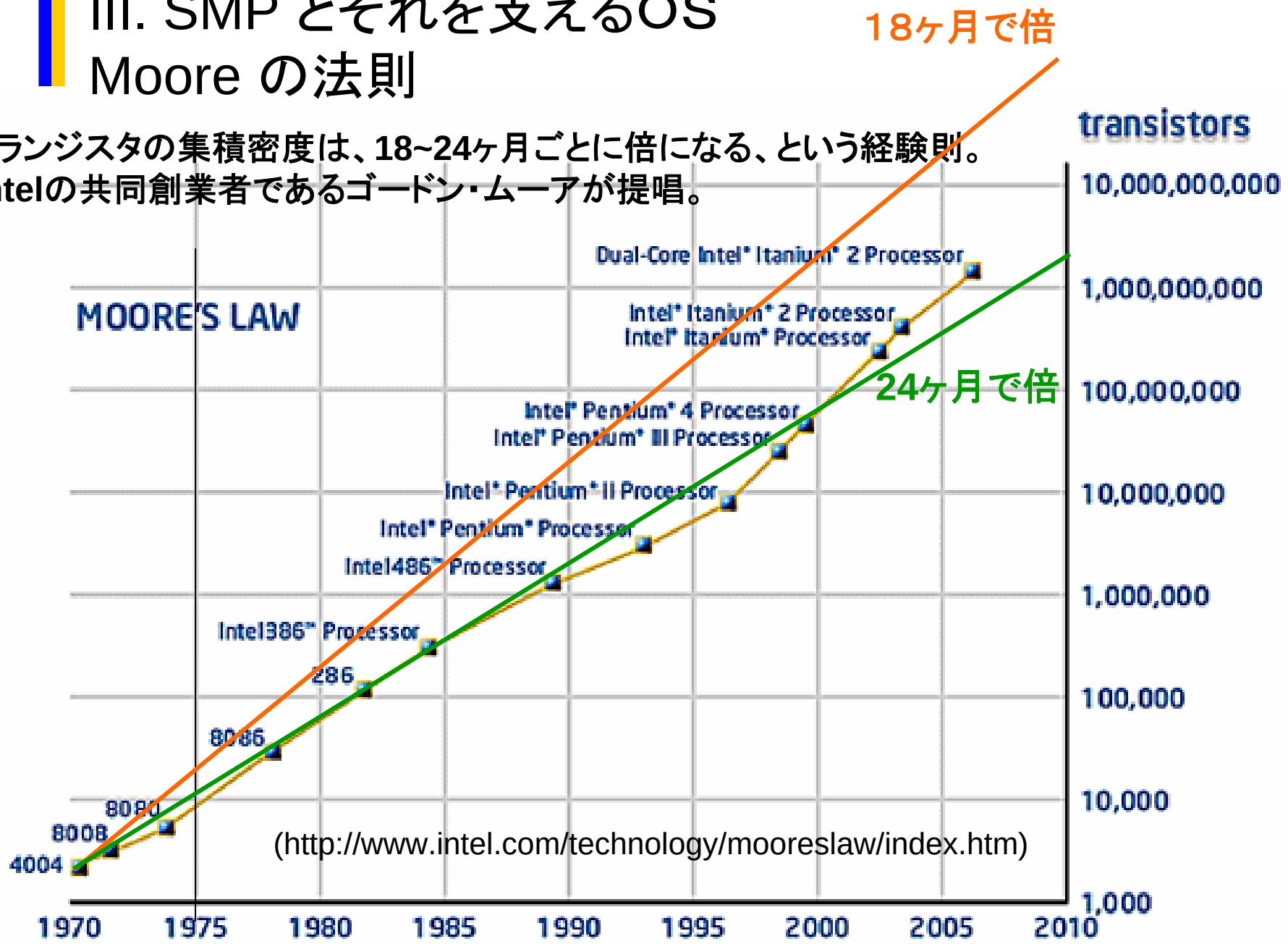
4. ロックの粒度およびスレッド数の問題
   - IO Completion Port (節6.1) などのデザインパターンを応用することにより、同期処理に比べて、非常に少ないスレッド（実CPU数に対応）で、必要な応答性を確保できる。したがって、スレッド競合による性能劣化が発生しにくい。

5. 非決定性

- 必要なスレッド数が少なく、ディスパッチ戦略により非決定性があらかじめシステムが定めた範囲に制限されるので、比較的容易。
- システムの状態は、非同期処理の処理待ち要求キュー、処理中要求表、スレッドのイベントキューなどで明示的に表現される。
- 要求のキャンセルは、処理待ちキューや処理中要求表などへの操作で明示的に実行可能。
- スレッド向けシグナル、ノンブロッキングI／O、リアルタイムシグナルなどの機能を新たに導入する必要はない。

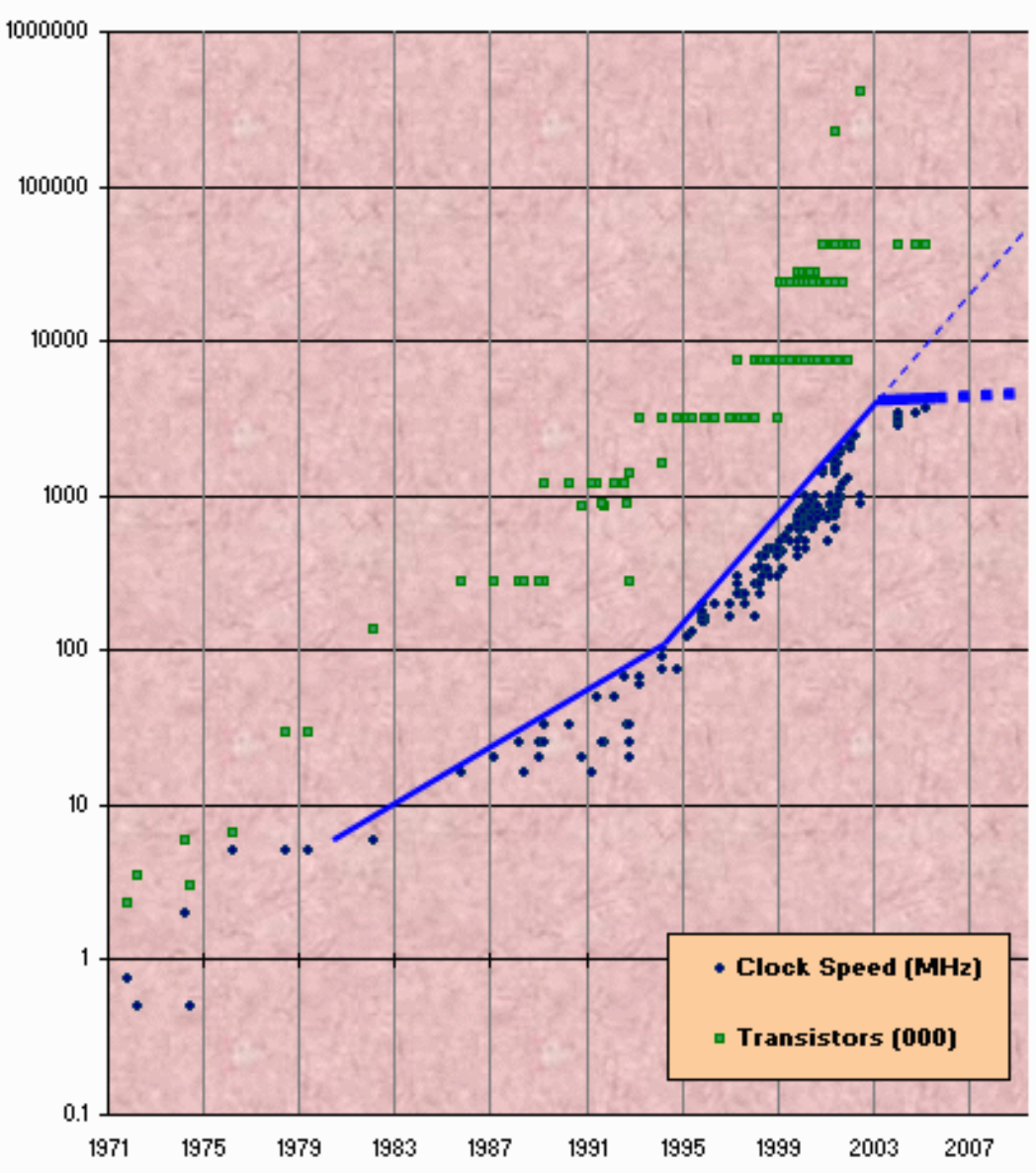
III. SMP とそれを支えるOS
Moore の法則

トランジスタの集積密度は、18~24ヶ月ごとに倍になる、という経験則。
Intelの共同創業者であるゴードン・ムーアが提唱。



(http://www.intel.com/technology/mooreslaw/index.htm)

トランジスタ数の伸びとクロックの乖離



(Intel: Wikipedia)

CPUに関する動向

- CPUクロックの伸びは、５年間で5倍程度に延びてきたが、２００５年を境界に、飽和している。
- CPUは、クロックアップではなく、キャッシュの増大とCPUコア数の増加で、速度を上げようとしている。

- キャッシュの増加は、メモリ速度が、プロセッサ速度ほど向上していないことも大きな理由になっている。

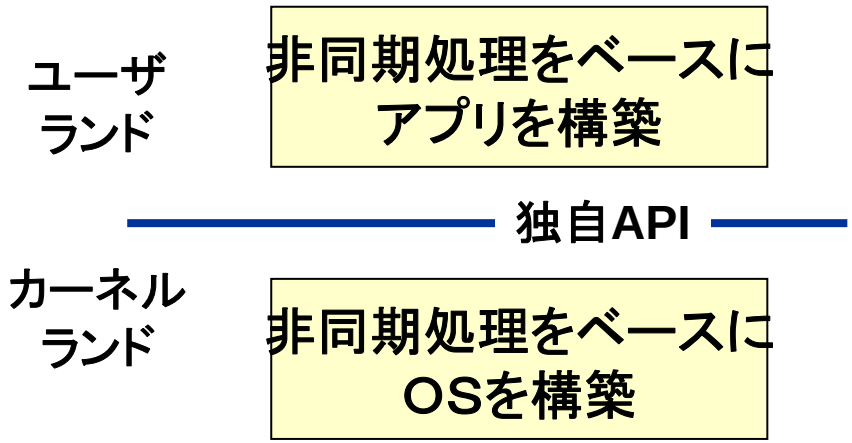
主要なマルチコアプロセッサの電圧とクロック

プロセッサ名称			クロック	コア電圧	リリース
Desk Top (Quad Core)					
Core 2 Extreme (Quad)		**TDP 130W**	3.0GHz	1.1 - 1.372V	2007.7
Core 2 Quad		**TDP 95W**	2.667GHz	1.1 - 1.372V	2007.7
Notebook (Dual Core)					
Core 2 Extreme		**TDP 44W**	2.8GHz	1.0375 - 1.3 V	2006.7
		TDP 35W	2.4GHz	1.0375 - 1.3 V	2007.5
Core 2 Duo	LV	**TDP 17W**	1.6GHz	0.9 - 1.1 V	2007.5
	ULV	**TDP 10W**	1.2GHz	0.750 - 0.925 V	2007.4

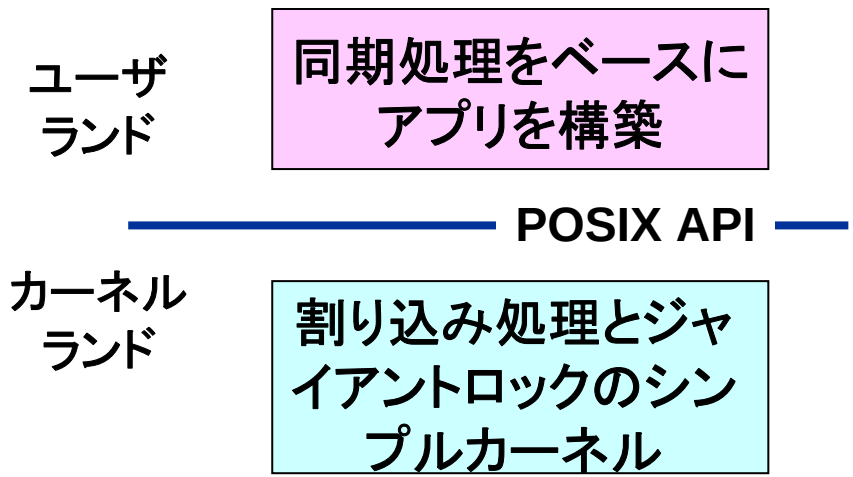
- 消費電力を下げるには、クロックを下げざるをえない。
- 絶対性能を稼ぐために、マルチコア化を推進。

Unix と Cutler’s Kernel

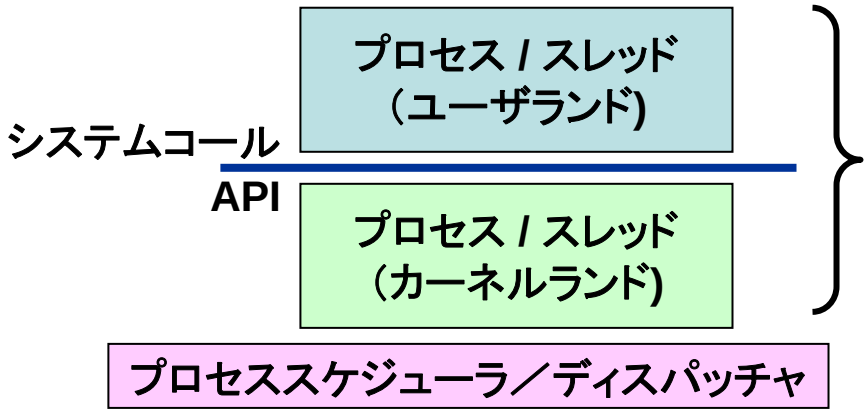
RSX-11M, VAX/VMS の構成



Unix の構成



IV 現在のOS階層の基本モデル



スケジューラブル エンティティ (SE)

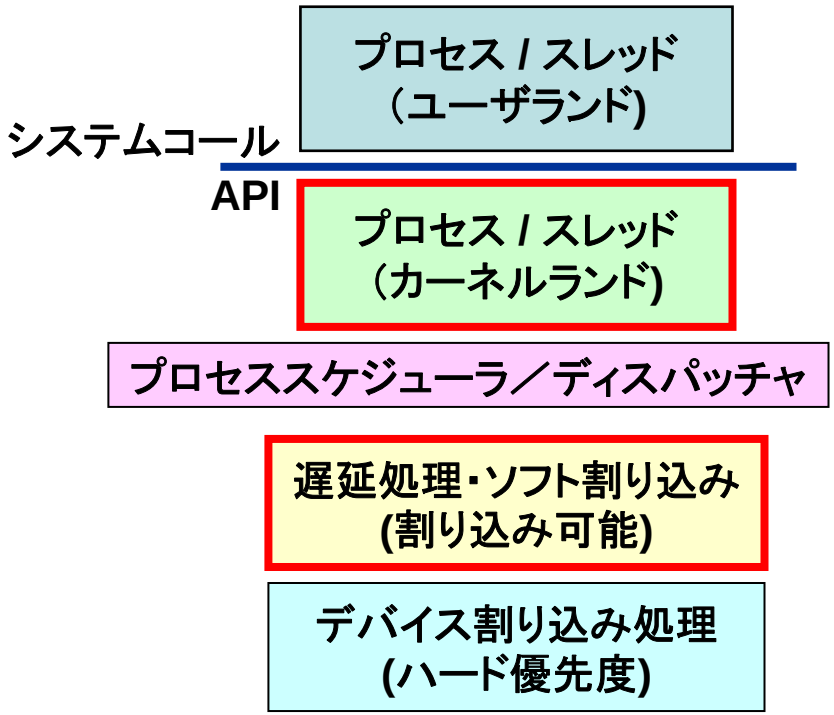
優先度、デッドラインに基づきスケジュールし、ディスパッチする。

ノンスケジューラブル エンティティ (NSE)

割り込みなどの、外部イベントに呼応して、ディスパッチされる。

- 上記構成のOSでは、一般にNSEの処理を、SEの処理に対して優先する。
- したがって、NSE で優先度の低いプロセスのための処理をすると、結果として、優先度の高いSEの処理が遅延する優先度逆転が発生しやすい。
- リアルタイムOSでは、遅延処理・ソフト割り込みの部分を縮小ないしなくし、SEに処理を移行させようとする場合が多い。

モノリシックカーネル (古典的) OSモデル



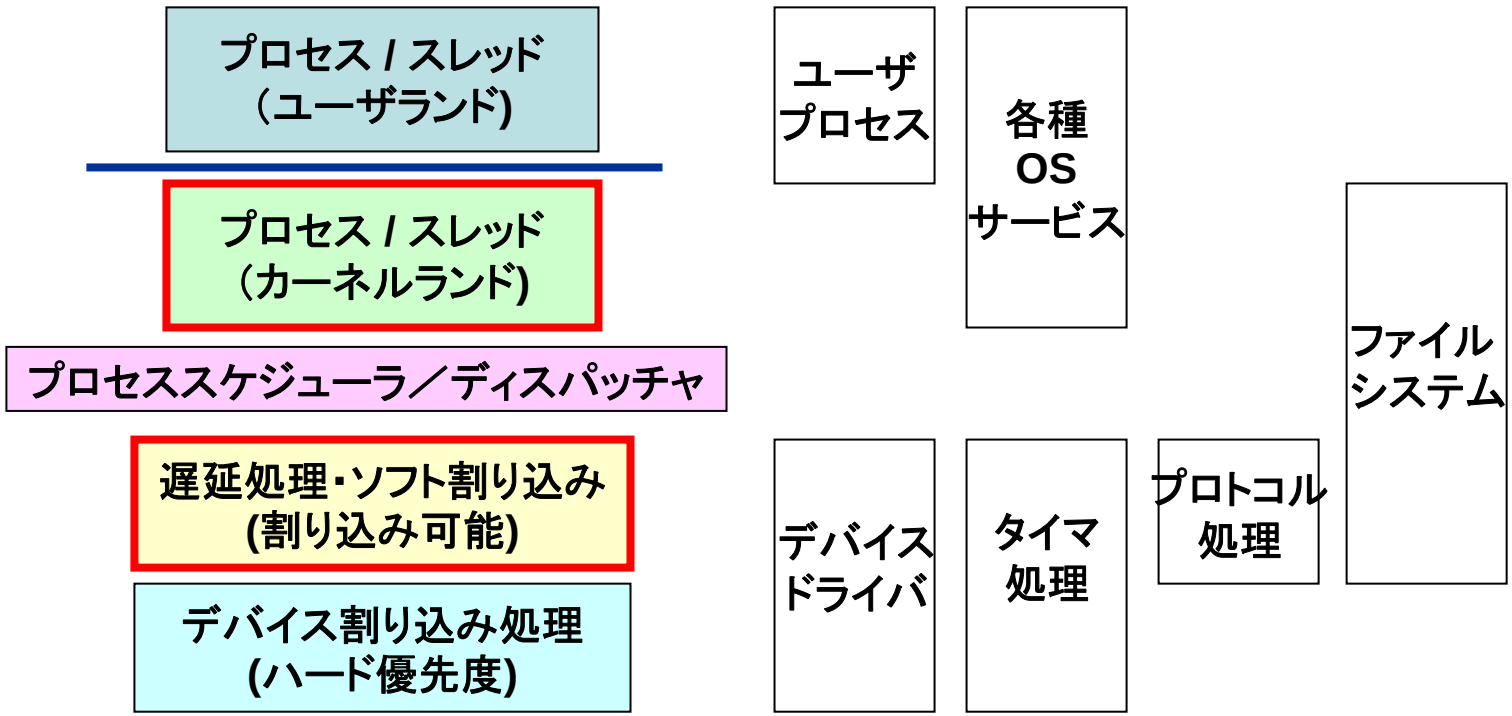
あまりタイムクリティカルでない、比較的間歇的かつ長期間の継続的制御を実装。

ユーザスレッドと優先度制御すべき時間のかかる処理を実装。

ドライバの制御部（BH: Bottom Half) や、プロトコル処理、ファイル処理など、カーネル内で高速かつ円滑にデータを移動させる処理を実装。

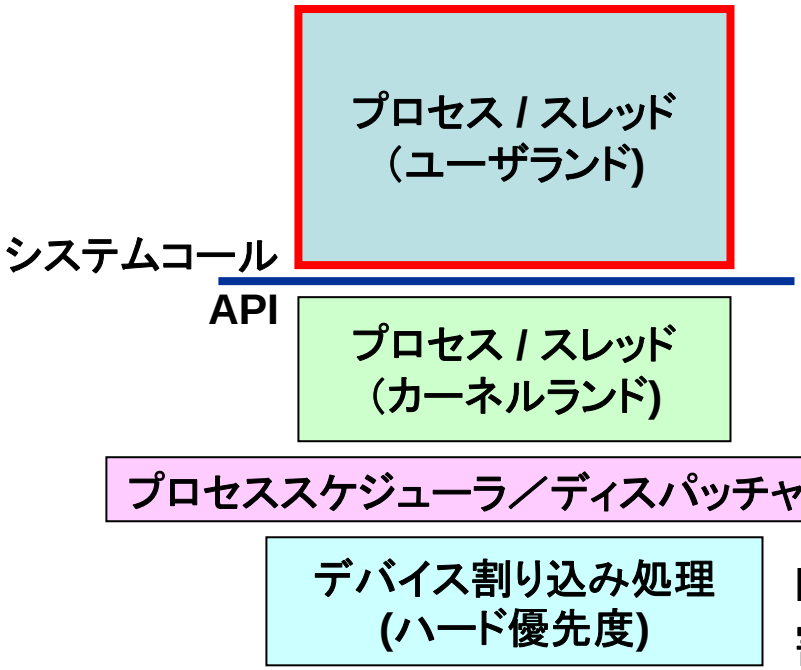
- OSリアルタイム機能の大部分をカーネルスレッドと遅延処理・ソフト割り込みで実装。
- OSとしては、カーネルランド3階、ユーザランド1階の4階建て
- ユニプロセッサでは、低優先度プロセスのためのカーネルサービスがNSEになると、優先度逆転のリスクが高まる。

モノリシックカーネル (古典的) OSモデル （つづく）



- カーネルが機能別にモジュール化されている点では、次のマイクロカーネルと大きな差はない。
- カーネルスレッドを用いたモジュールを作成すると、高優先度のSEと、処理に時間がかかるOSサービスとを、スレッド優先度で調整することが可能になる。
- OSサービスのデータの流れは、主にカーネルランド内で垂直方向になる。
- 処理スレッドをデータと一緒にモジュール間にわたって動かせるので、処理効率を確保しやすい。

マイクロカーネルOSモデル



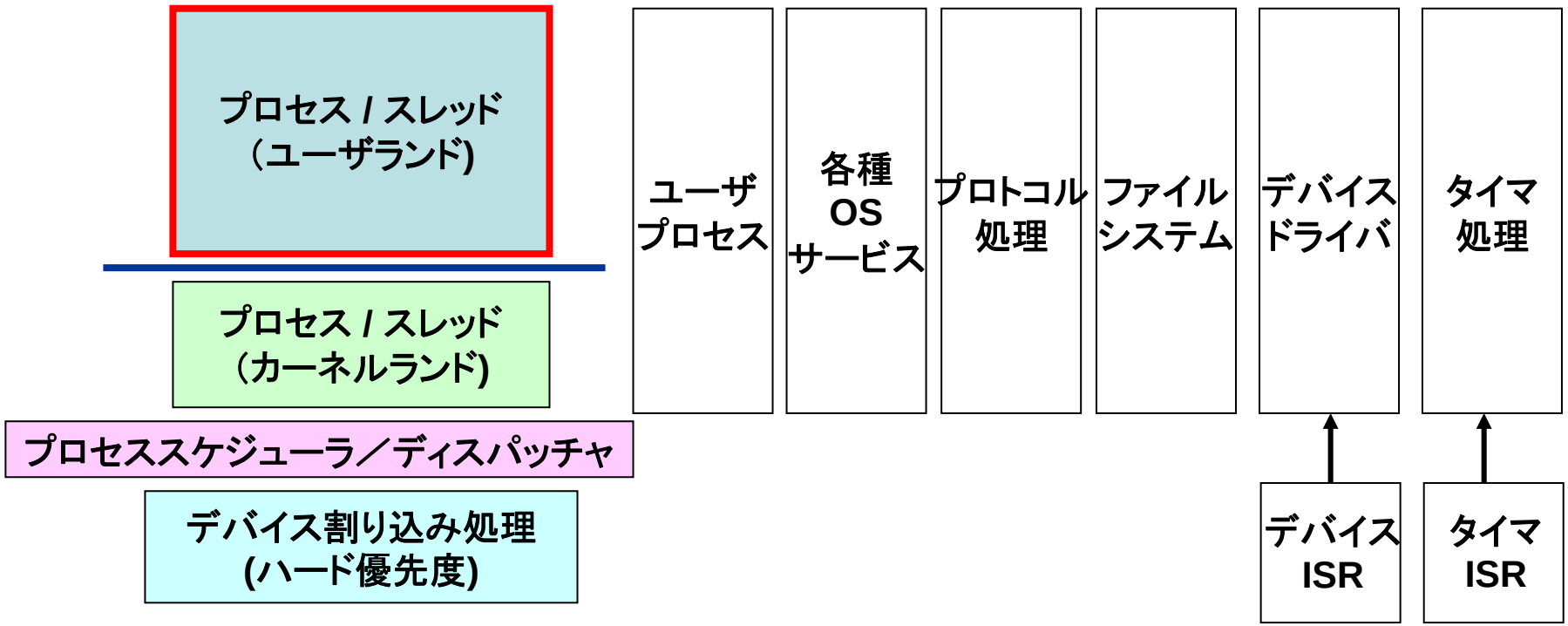
OSサービスを含めて、
大部分の処理をユーザランドの
プロセスに持ち込む。

- 高度なスケジューラビリティの確保。
- バグへの安全性の向上

NSEの最小化
割り込み処理も例外ではない。

- OS機能の大部分を、ユーザランドのプロセスで実行。
- カーネルは、メモリ管理、プロセス管理、アクセス制御などの基本機能に限定。
- NSEの処理を極小化することで、NSEの影響による優先度逆転を回避。

マイクロカーネルOSモデル（つづく）

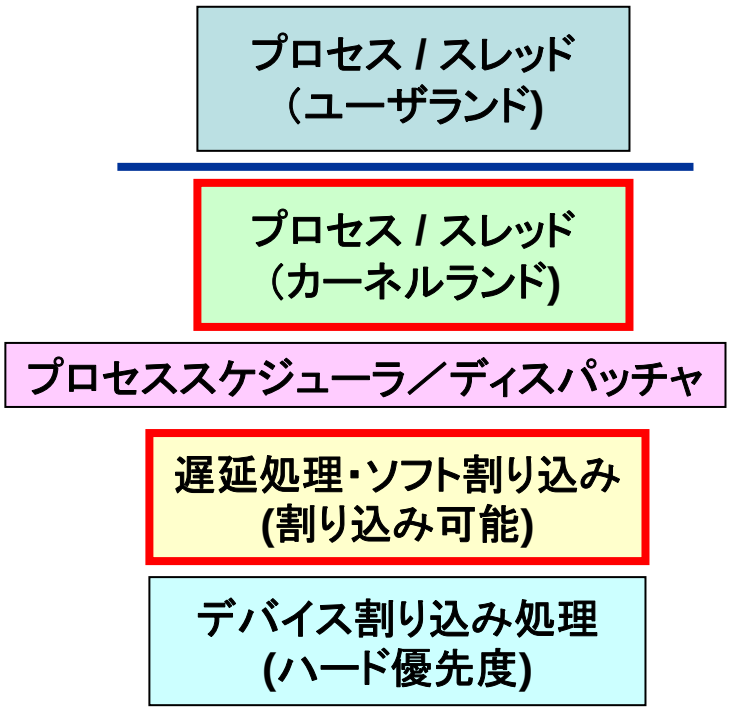


- カーネルサービスの大部分がSEで実行されるので、優先度制御が容易。
- OSサービスのデータの流れは、主にユーザランド内で、水平方向になる。
  この際、プロセスの境界をまたぐので、コンテキストスイッチが発生する。

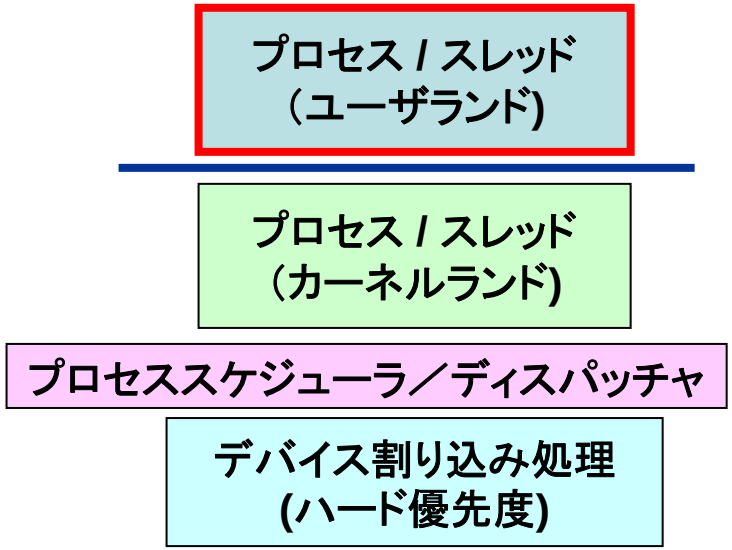
ISR: Interrupt Service Routine 割り込み処理ルーチン

リアルタイムOS／プログラム

(1) モノリシックカーネル型RT処理派
(データ・制御一体派)



(2) マイクロカーネル型RT処理派
(スケジューラビリティ優先派)



リアルタイム処理の要件

1. リアルタイム制約の満足 （SE，NSEそれぞれ）
2. 実行効率
3. 記述の容易性
4. モジュール合成の柔軟性
5. バグへの耐性
6. SMPへのスケーラビリティ
7. 負荷へのスケーラビリティ
8. 計算のキャンセルや電源異常処理などの信頼性

V. David Cutler の処理モデル

1. 3階と4階にあるプロセスも含めて、機能実装にあたり、**非同期性**を重視。非同期APIを多用することで、スケーラビリティと柔軟性があり、かつ信頼できるシステムを設計・実装できるようにする。
2. カーネルでは、非同期な処理開始にあたり、スレッド全体ではなく、その一部の処理コンテキストを選択的に差し替える**多重コンテキスト**を実装する。

   このコンテキストは、同一CPUにおいて、優先度の低いものから高いものへ、スタック可能である。
3. 排他制御は、あくまでデータを対象に行うことが大原則。したがって、モジュールの境界での逐次化を行わない（**非逐次化**）。

5.1 非同期処理

複数の要求を並行して実行可能。
要求を出すスレッドと、結果を処理するスレッドを分離できる。

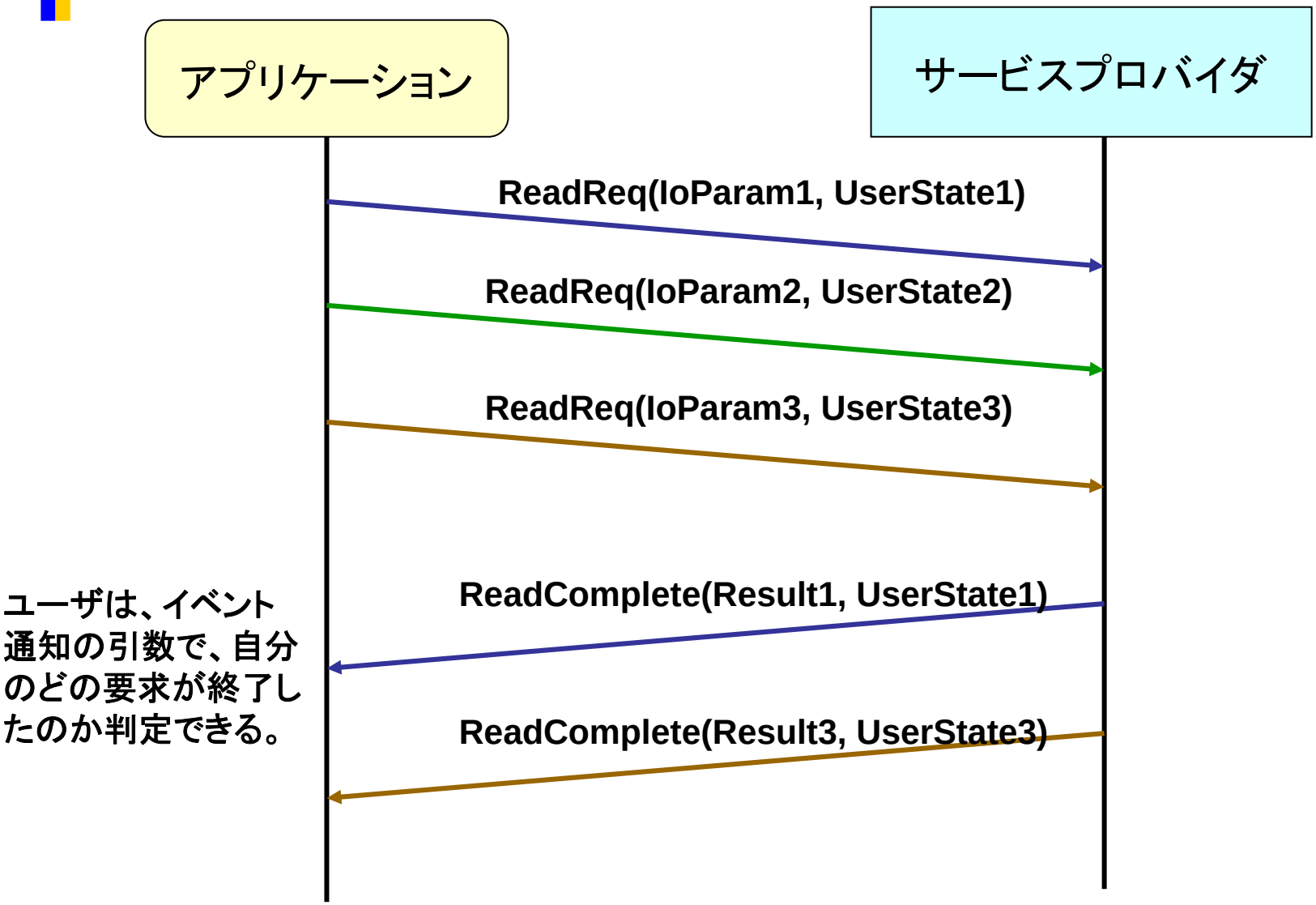
要求処理時:

要求それぞれを区別するための識別情報を、サービス要求時に、サービス用パラメータといっしょに、引数として渡す。

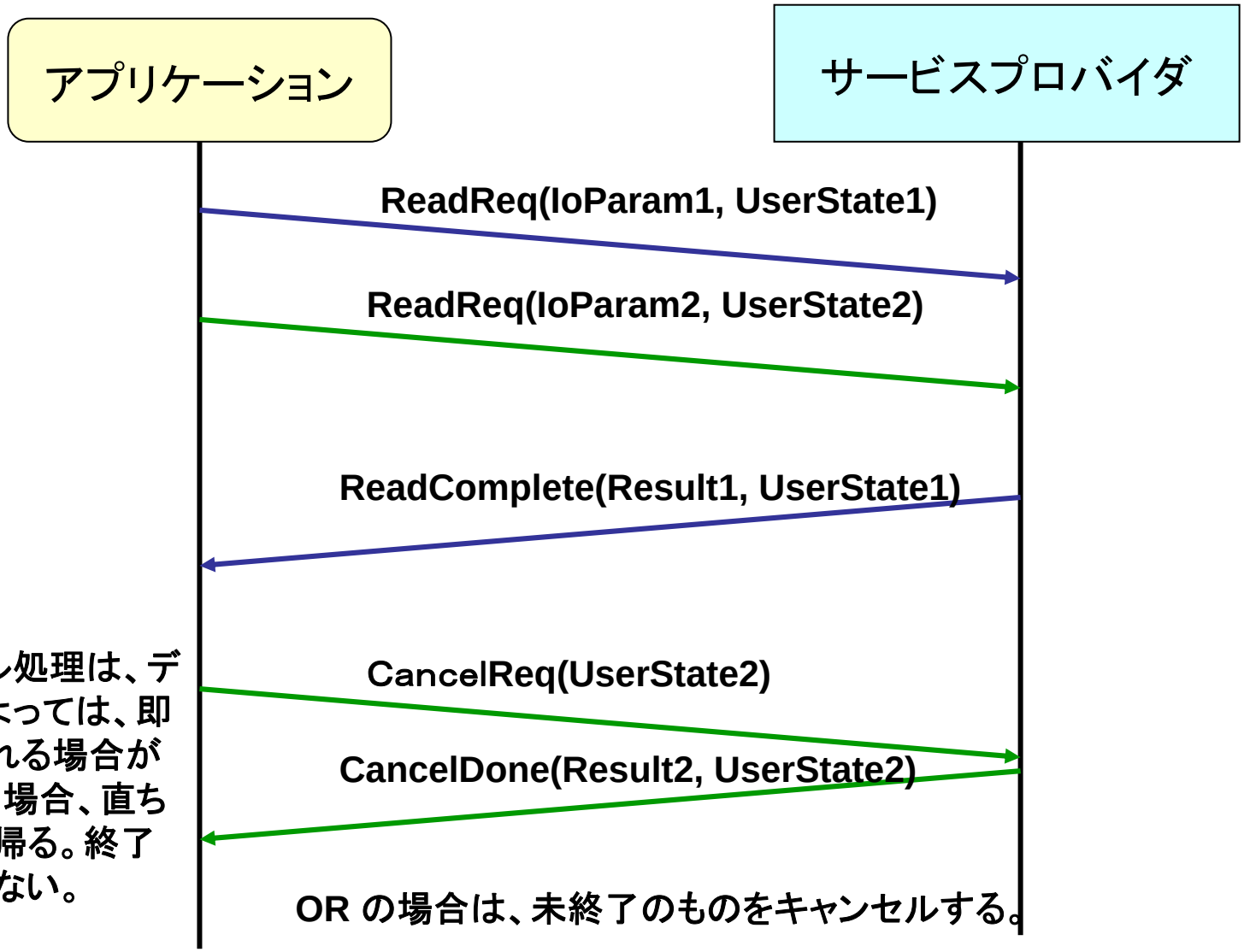
処理終了通知時:

終了した処理に対応した識別情報を、サービス結果通知に加えて、指定先に通知する。この通知は、スレッド対応にキューイングされ、イベント、あるいはコールバックにより処理される。

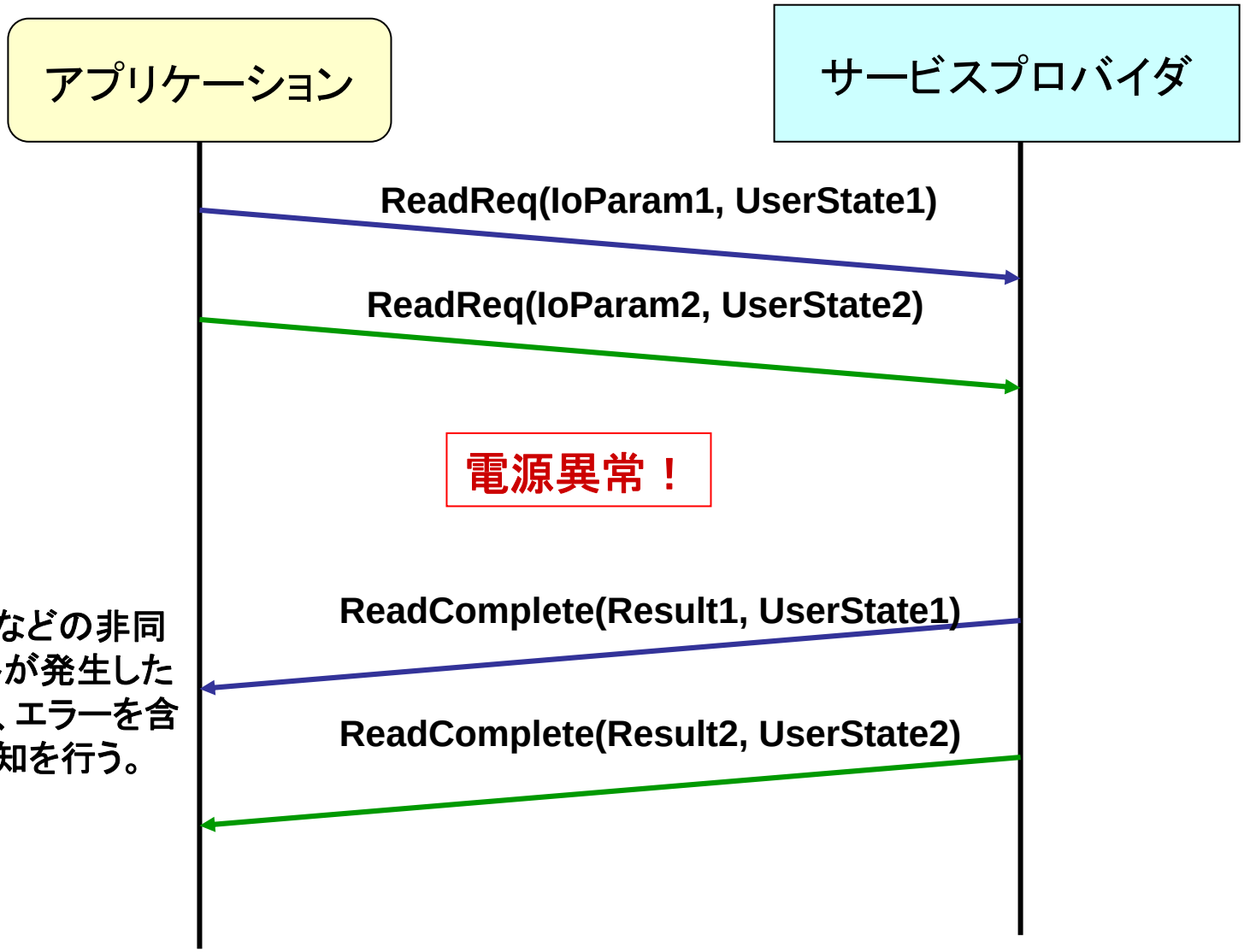
非同期処理：複数の要求を並行して実行 （つづく）



非同期処理：未終了要求の(自発的)キャンセル



非同期処理：未終了要求の強制的キャンセル



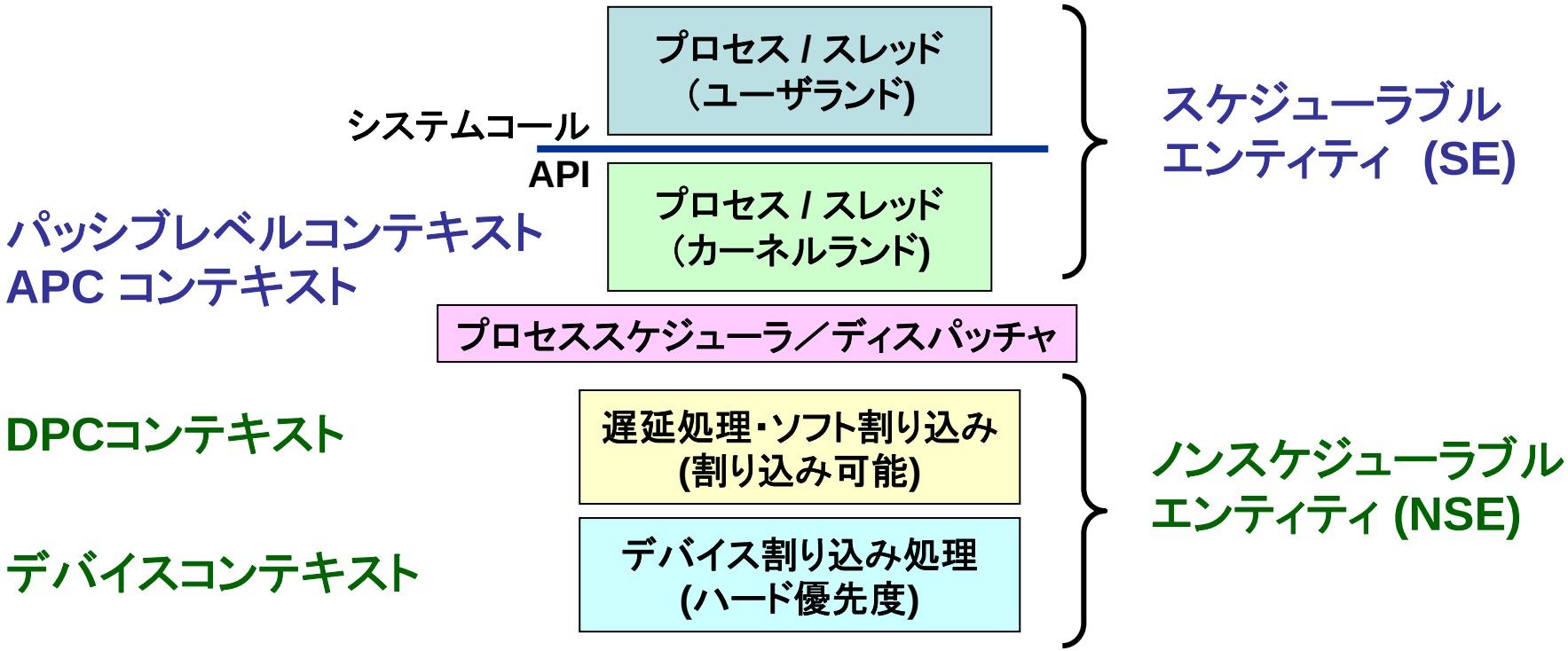
電源異常などの非同期イベントが発生した場合には、エラーを含む完了通知を行う。

5.2 多重コンテキスト

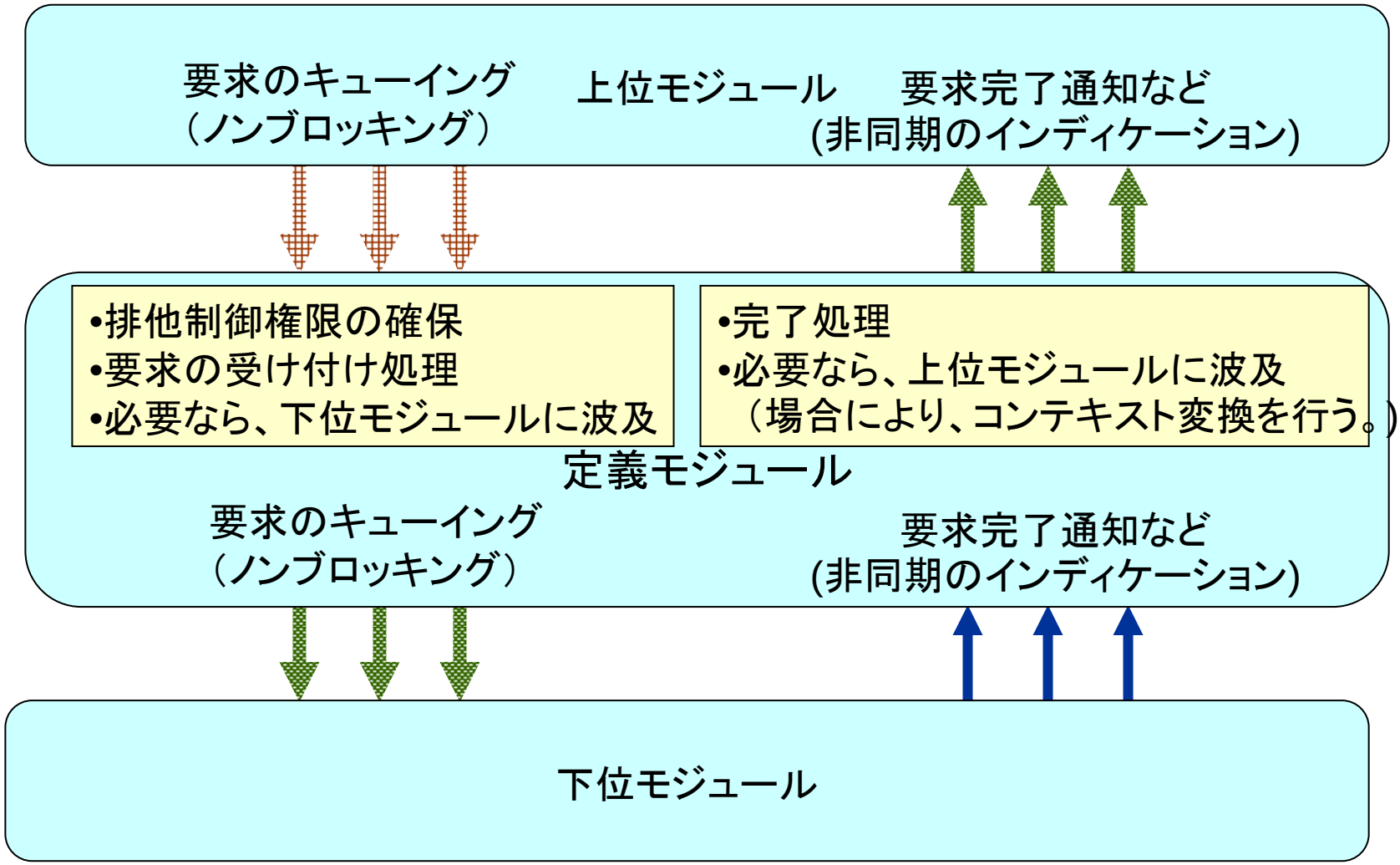
- 名前の呼び方は、異なるが、リアルタイムOSはおおむね、以下の処理コンテキストをもつ。コンテキストとは、処理開始時のシステム全体の一貫性の度合い(ないし非同期の度合い）の程度を表している。
- カーネルサービスは、呼び出し可能なコンテキストが指定される。それ以外のコンテキストから呼び出すことは禁止される。
  (カーネルの各オブジェクトのメソッドには、呼び出し可能なコンテキストが指定される。）
- 適合したコンテキストからは、エントリを排他制御ロックなしに直接呼び出せる。
- コンテキストが適合しない場合、間接的に別のコンテキストのタスクを起動し、それに処理を代行させる。これは、Proxy 処理とみなせる。これにより、必要な処理を適切なコンテキストで行わせることが可能になる。
- ユーザ権限でのアクセス制御が必要なエントリは、ユーザスレッドのコンテキストも指定される。

- コンテキストには、実行するCPUに紐付けられたCPU固有コンテキストと、そうではない一般コンテキストの2種類がある。
- CPU固有コンテキストは、NSEに対して適用され、呼び出しは、CPUごとに逐次化されてディスパッチされる。排他制御や同期でのブロックは許されず、したがって、入出力要求などは、すべて非同期処理で行われる。
- CPU固有コンテキストでは、複数CPU間の逐次化は行われない。この排他制御には、スピンロックを用いる。
- 一般コンテキストは、SEに適用され、その排他制御には、Mutex, Semaphore などのブロック型のものを含む任意のものが利用可能である。

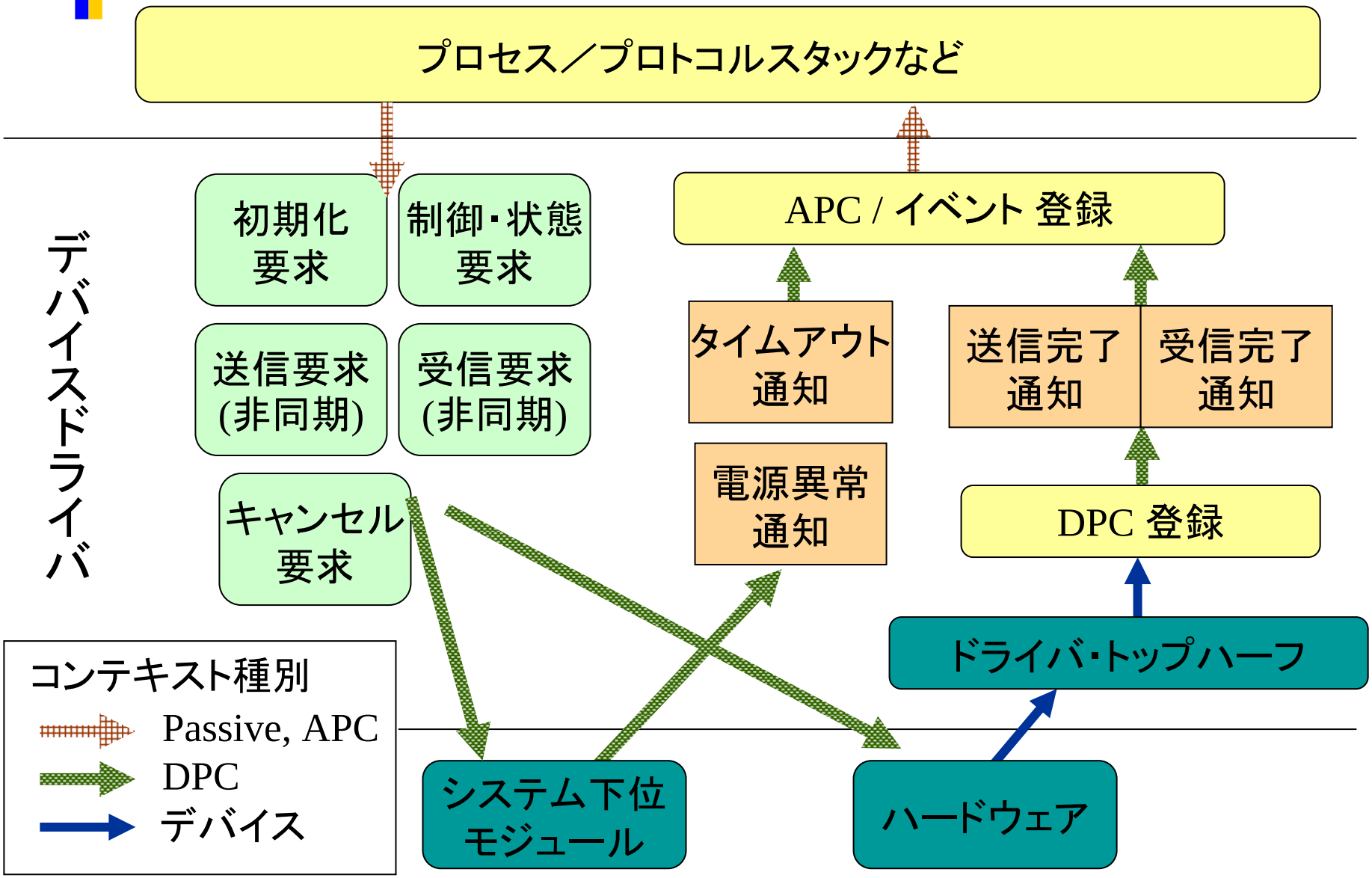
例： 現在のWindows OS階層の基本モデル



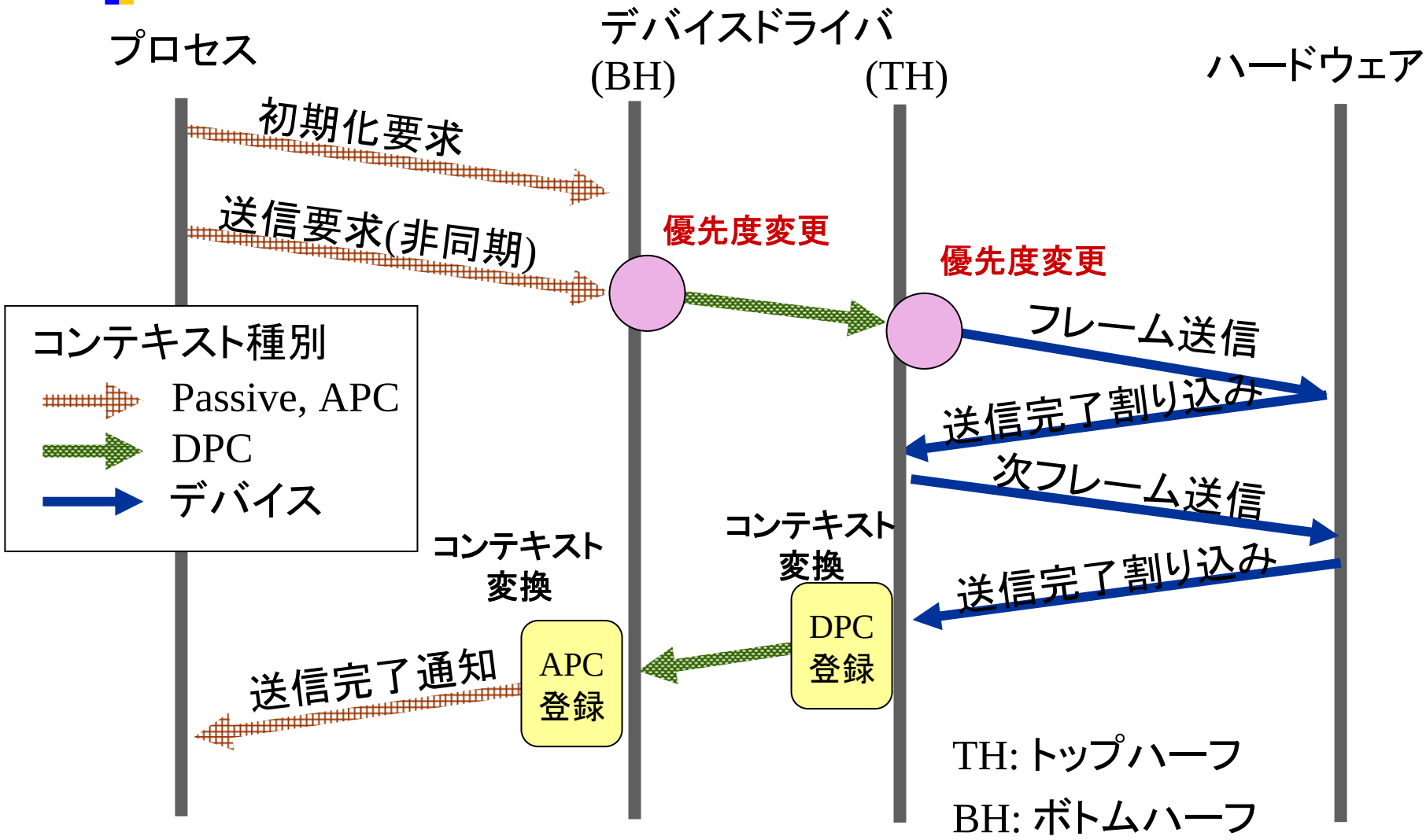
モジュール化された非同期処理構成



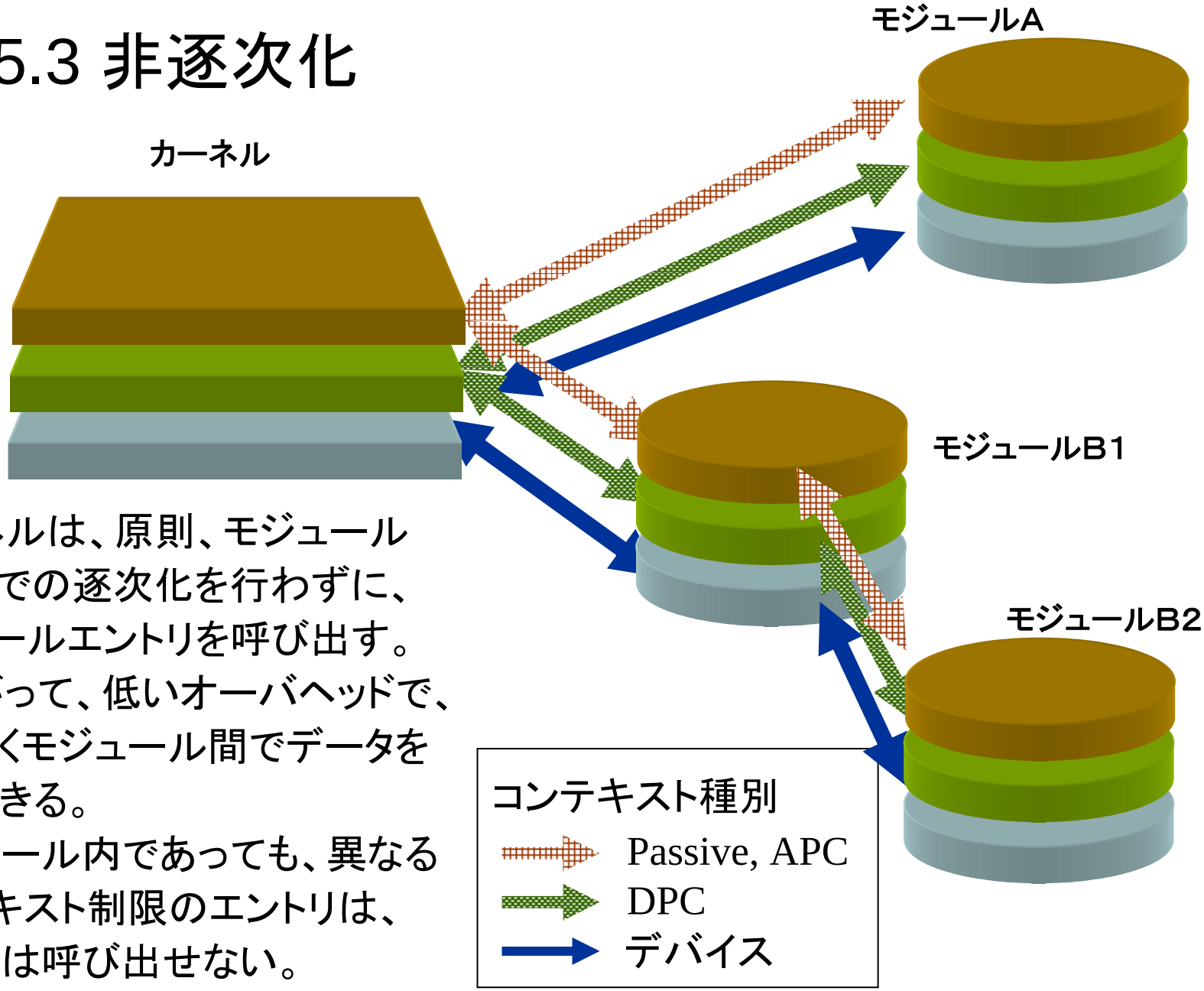
単純化された非同期通信モデルの例



各モジュールの動作シーケンス例



5.3 非逐次化

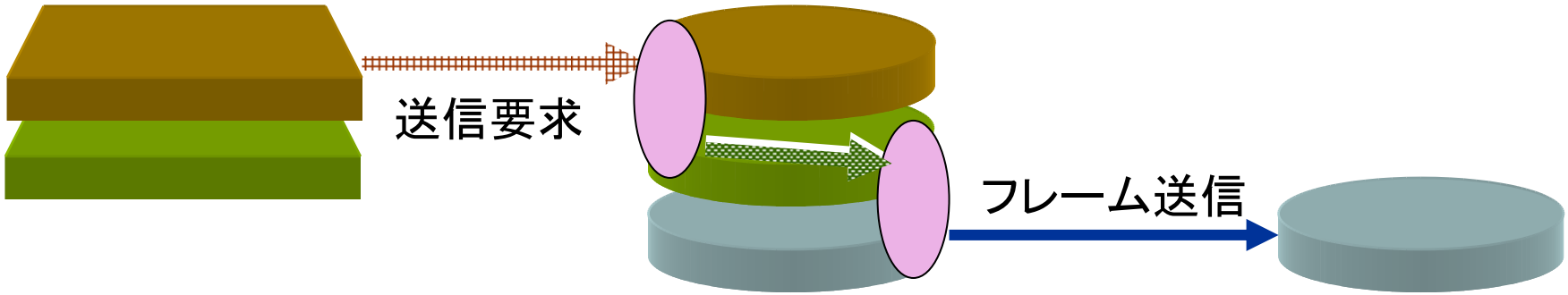


- カーネルは、原則、モジュールレベルでの逐次化を行わずに、モジュールエントリを呼び出す。
- したがって、低いオーバヘッドで、効率よくモジュール間でデータを授受できる。
- モジュール内であっても、異なるコンテキスト制限のエントリは、直接には呼び出せない。

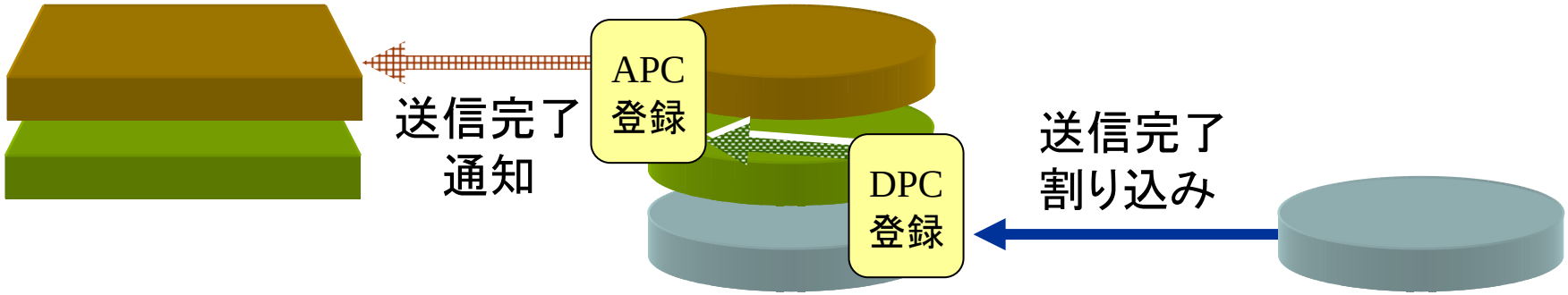
5.3 非逐次化

プロセス　　デバイスドライバ (BH) (TH)　　ハードウェア

(1) 送信要求の流れ



(2) 送信完了通知の流れ



- 非逐次化は、モジュール境界での不要なロックを削減するので、特にSMP化での性能改善に効果がある。ただし、もっともバグが出やすいところでもある。
- 特に、あるモジュールの手続きがロックを保持したまま、別のモジュールを呼び出し、そこから再び自分のモジュールが呼び出される**再突入**は、予想外のエラーを引き起こすことがあり、十分な注意が必要。

VI. デザインパターンの実例

1. **I/O Completion Port:** 数千のソケットをSMPの並列性を生かしながら、スケーラブルに処理。

2. **Receive Side Scaling:** 複数CPUでプロトコル処理することにより、高性能を達成。

3. **DPS: Dynamic Protocol Stacking**:
   動的バインドバスをもつ柔軟なプロトコル実装。

リアルタイム処理要件からみた Cutler モデル

1. リアルタイム制約の満足:

   プロセスベースのマイクロカーネルより、優先度逆転が発生しやすい。

   ソフトリアルタイムでは、実行効率のよさで、絶対性能的にはカバーしやすい。

2. 実行効率

   実CPU数に呼応した少ないスレッドで、効率的に処理

3. 記述の容易性

   状態遷移的な記述で、形式的仕様記述と適合性が高い

4. モジュール合成の柔軟性

   非同期処理は、スレッドとライブラリのような差がなく、合成が容易。

5. バグへの耐性

   プロセスベースのマイクロカーネルには劣る。

   モジュール性が高くコード理解が容易なので、バグ発生を少なくできる？

   (6.3 DPS 参照)

6. SMPへのスケーラビリティ　(6.2 RSS 参照)
7. 負荷へのスケーラビリティ (6.1 IOCP 参照)
8. 計算のキャンセルや電源異常処理などの信頼性

   状態が待ち行列などで明示的であり、優れている。

6.1 I/O Completion Port (IOCP)

同期処理では、複数ディスクリプタの入出力完了をポーリングする方法として、Select, Poll などがある。

多数ディスクリプタを対象にSelect を実行した場合に、性能上の問題が存在することは、よく知られている。多数の休止ソケットと少数の活発なソケットの通信をポーリングする際に、特に性能低下が発生しやすい。

- Select 開始時に、監視対象ディスクリプター覧をユーザランドからカーネルランドにコピーし、終了時には、逆方向にコピーし直す必要がある。
- Select 処理において、カーネルは、ポーリング対象のディスクリプタを知るために、配列の全要素をスキャンする。
- Select 終了後、ユーザプロセスは、入出力可能となったディスクリプタを知るために、配列の全要素をスキャンする。
- これを 入出力完了による Select 実行の度に繰り返す。

IOCP のアプローチ

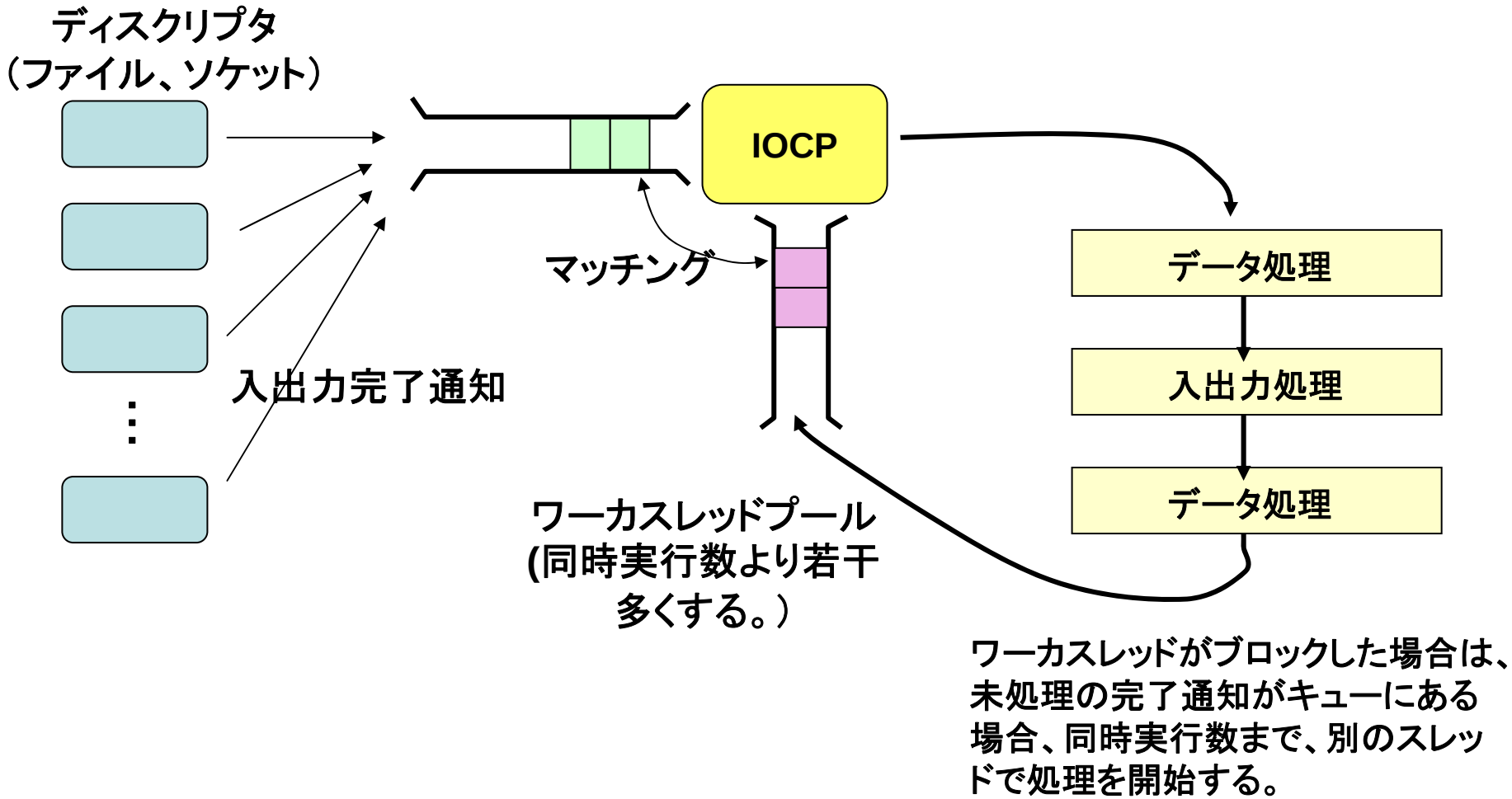
Windows NT 3.5 (1994) から実装

1. 入出力完了ポートという構造に、複数のディスクリプタを関連づけ、その入出力完了通知を多重化してキューイングできるようにする。
2. 入出力完了ポートには、ワーカスレッドを複数貼り付けることができる。この際、同時実行数を指定できる。
3. 入出力完了ポートに完了通知がなされると、ワーカスレッドがそれを取り出して、処理を開始する。
4. ワーカスレッドがファイルI／Oなどでブロックした場合は、それをカーネルが検知して、同時処理数まで別スレッドでの処理を開始する。

効率化の理由：

- 上記1の関連づけは、最初に行うだけでよい。
- 入出力完了通知は、それ自体で、どのディスクリプタのどの要求が完了したかが、わかるようになっているので、ディスクリプタをスキャンする必要がない。
- 複数のワーカスレッドを利用した並行処理が可能。

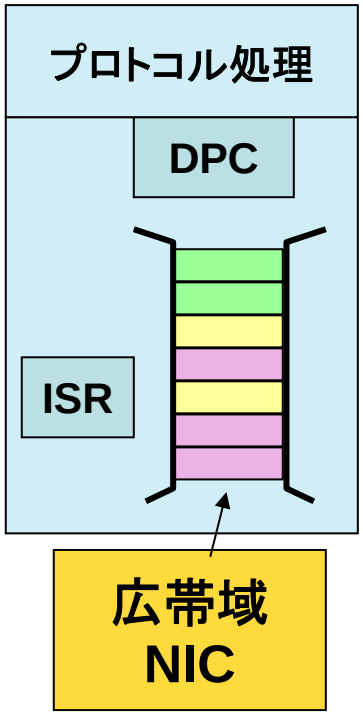
IOCP のアプローチ （つづく)



6.2 Receive Side Scaling (RSS)

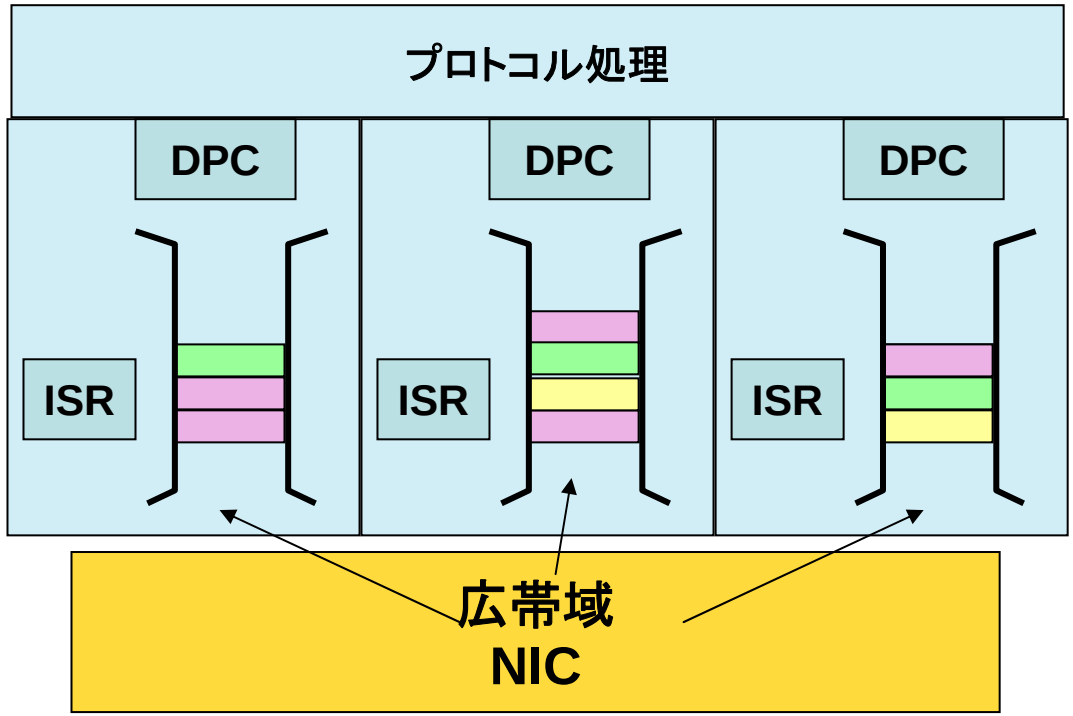
- 複数CPUでプロトコル処理することにより、高性能を達成。

(1) 1台のCPUで処理



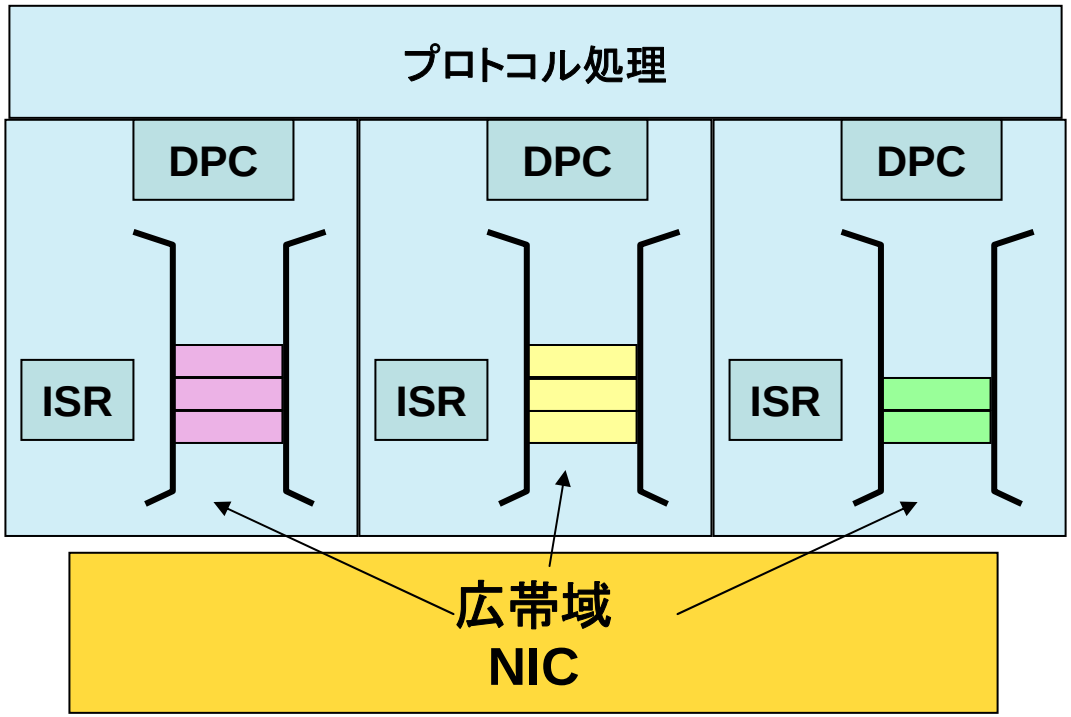
パケットリオーダなし。
SMPが生かせない。

(2) 複数のCPUにフレームをランダムに分散



パケットリオーダが頻発して、プロトコル処理のオーバヘッドが増大する。

RSS: NIC とプロセッサの連携により、フロー単位で、
フレームをプロセッサに分散させる。

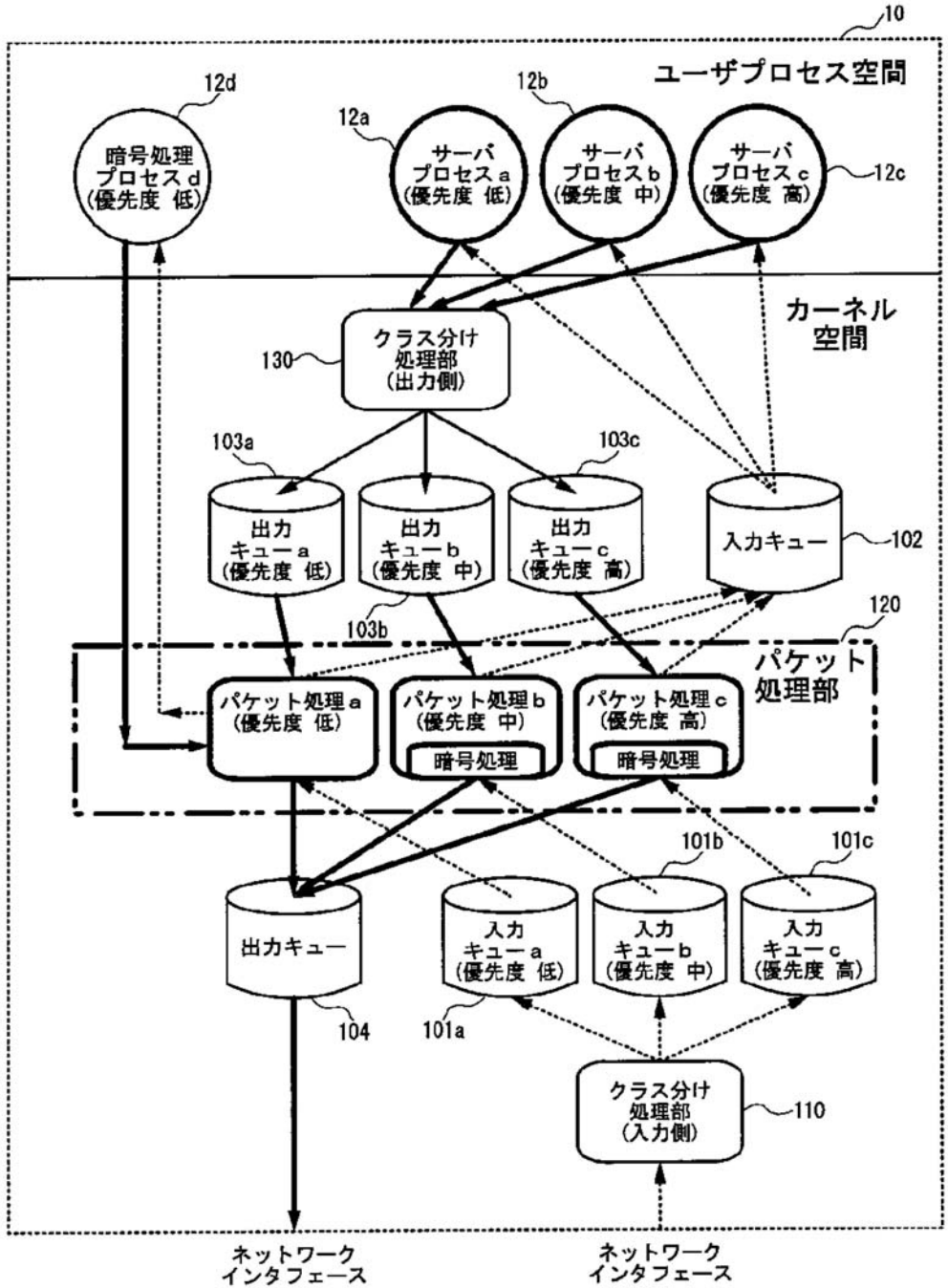


パケットのリオーダを発生させずに、プロトコル処理を
複数CPUに分散させることができる。

6.3 動的組み込みIPSec スタック（QoS制御機能つき)

Cutler モデルに基づき設計。
Solaris X86 SMPに実装

本庄・小野： 暗号化通信におけるサービス品質制御方法及び装置、
サービス品質制御プログラムを格納した記録媒体,
特開2001-344228, 特許 3736293
(2000.1出願、2001.12公開、2005.11登録)



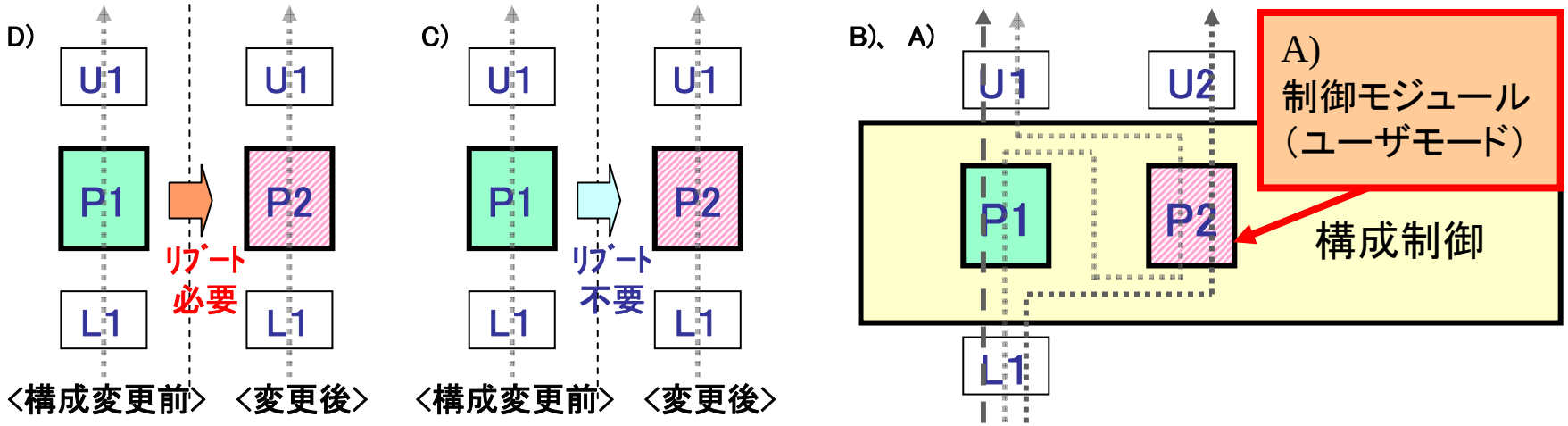
6.4 DPS: Dynamic Protocol Stacking: Linux での動的バインドバスをもつプロトコル実装

- Linux 上で、Nw. ミドルウェアを実装するため、カーネル内で、動的にプロトコルスタックを組み立てるDPS技術を開発し、実装。
- Linux カーネルを Cutler の方法でモデル化し、必要な場所にコンテキスト変換 Proxy を置いて実装。
- DPS 全体は、Linux の BH コンテキストで動作する。
- Linux カーネルにはローダブルモジュールとして組み込む。
- DPS の構成制御は、ユーザモードから専用API により行う。
- DPSで構築されたスタックは、Linux カーネルには、仮想イーサとして見せることが可能。
- 各スタックコンポーネントは、すべて非同期モジュールになっている。
- BSD系のものを改造して利用。

動的なバインドパスの構築に向けたモデル

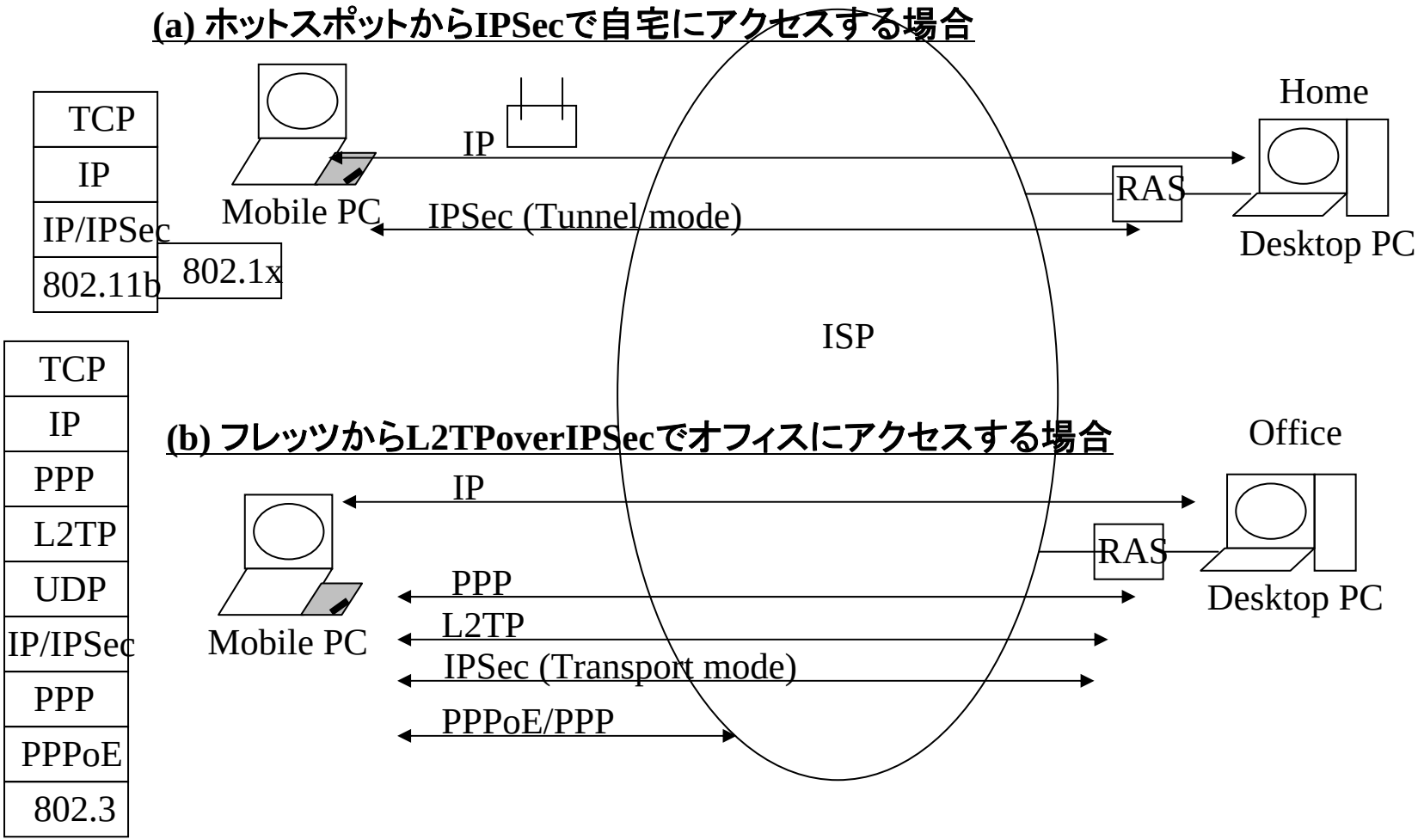
	:OS制御・静的 バインドパスモデル	:OS制御・動的 バインドパスモデル	プラットフォーム制御・動的 バインドパスモデル (OS依存形ﾄﾞﾗｲﾊﾞ)
ﾊﾞｲﾝﾄﾞの 自由度	・バインド対象はOSに よって定められる.	・バインド対象はOSに よって定められる.	・本プラットフォームで バインド対象を選択できる.
ﾊﾞｲﾝﾄﾞの 動的構成変更	・バインド機構は静的で システム動作中に 変更できない (リブートが必要)	・バインド機構は動的で システム動作中にも 変更可能(リブート不要)	⇦ (同左)
実装例	・Windows98, Me	・Windows2000, XP	Linux 版 DPS 技術

高度化のレベル →



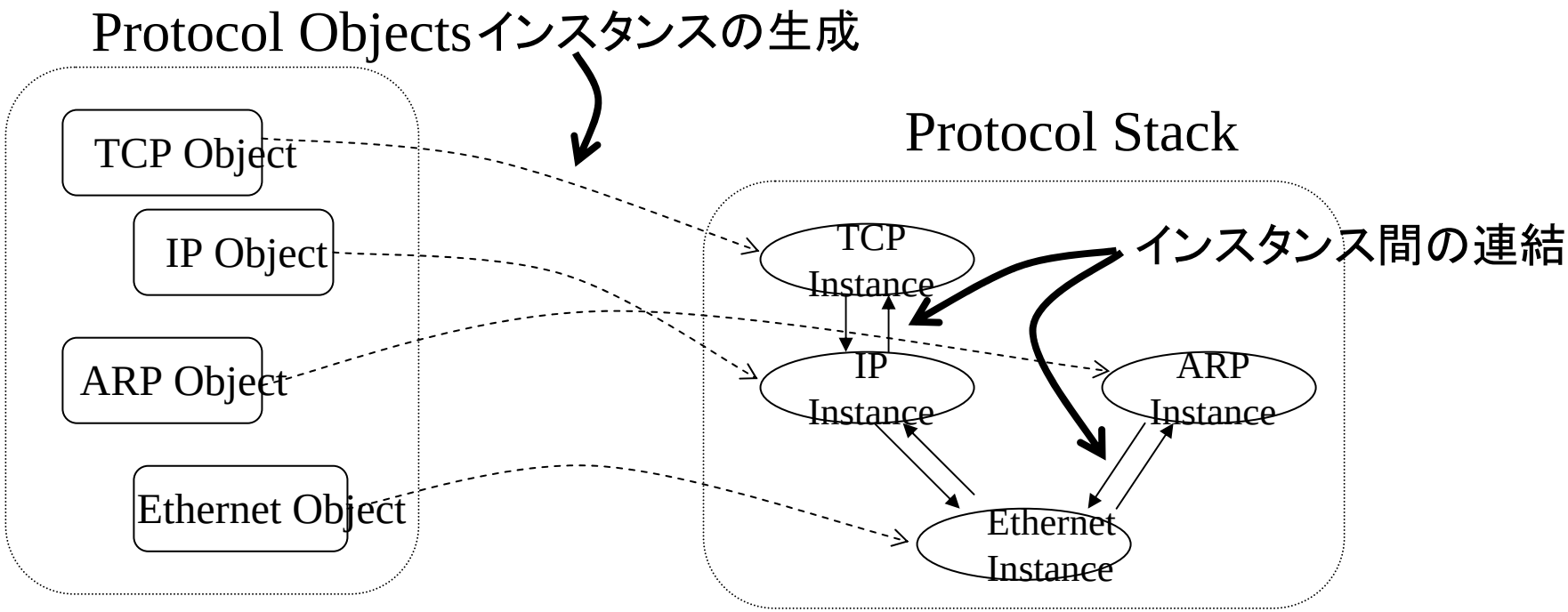
ノマディックプロトコルスタック

接続環境、接続先に応じて、必要なプロトコルを組合わせてプロトコルスタックを動的に生成する枠組み

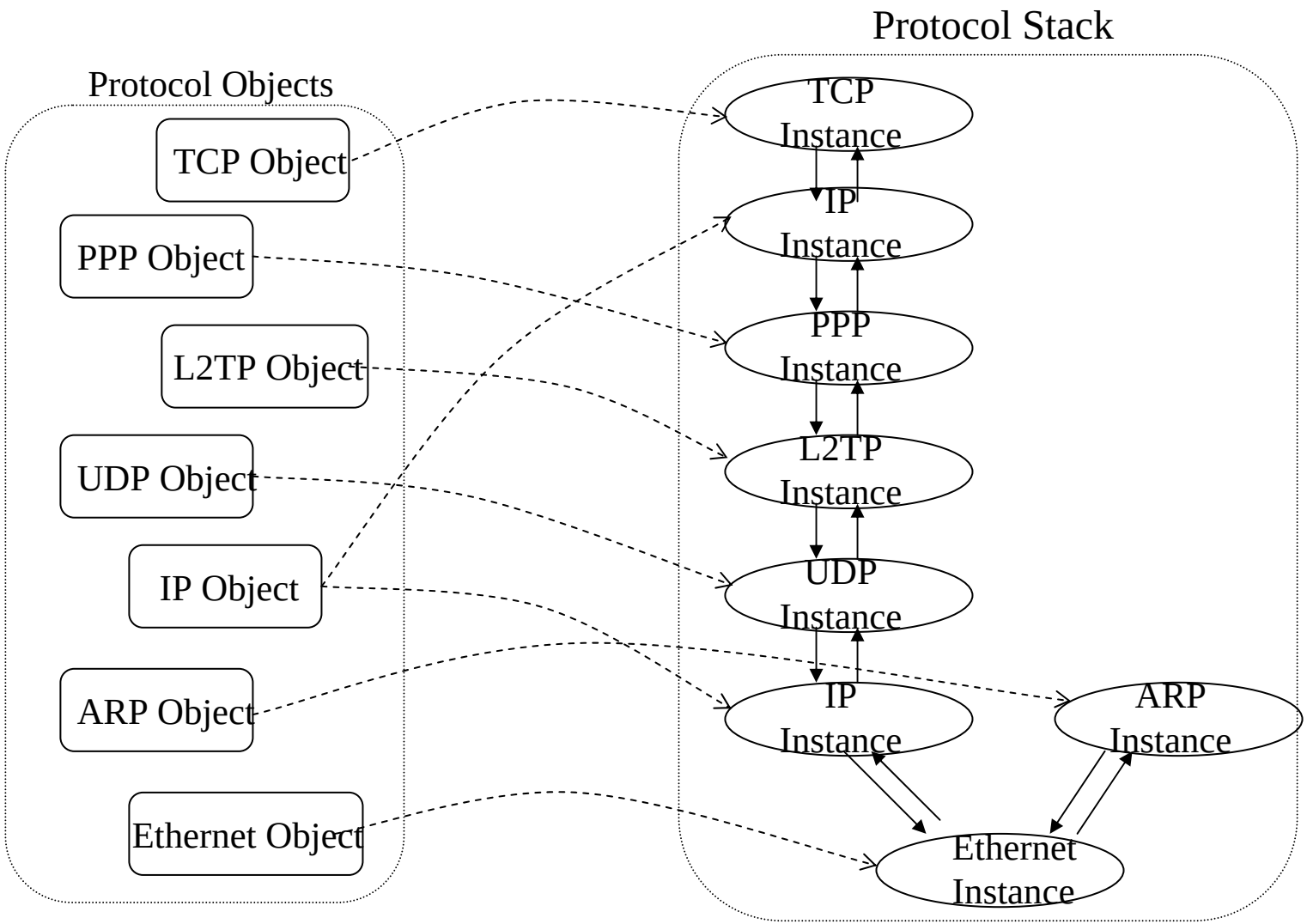


ノマディックプロトコルスタックの実現方法

- 各プロトコル(IP、TCP、PPPoE、PPPなど)をオブジェクトとして用意。
- オブジェクトインスタンスを動的に生成
- インスタンス間を連結することによって、プロトコルスタックを構成。



ノマディックプロトコルスタック生成例



プロトコルオブジェクト/インスタンスのメソッド

各プロトコルオブジェクト、プロトコルインスタンスに対して、以下のメソッドを用意することにより、プロトコルスタックの生成、削除を行う.

プロトコルオブジェクトに対するメソッド

プロトコルオブジェクトに対しては、インスタンスの生成(Create)、削除(Destroy)の2つのメソッドを与える.

- Create
  プロトコルインスタンスを生成する.
- Destroy
  プロトコルインスタンスを破棄する.

プロトコルインスタンスに対するメソッド

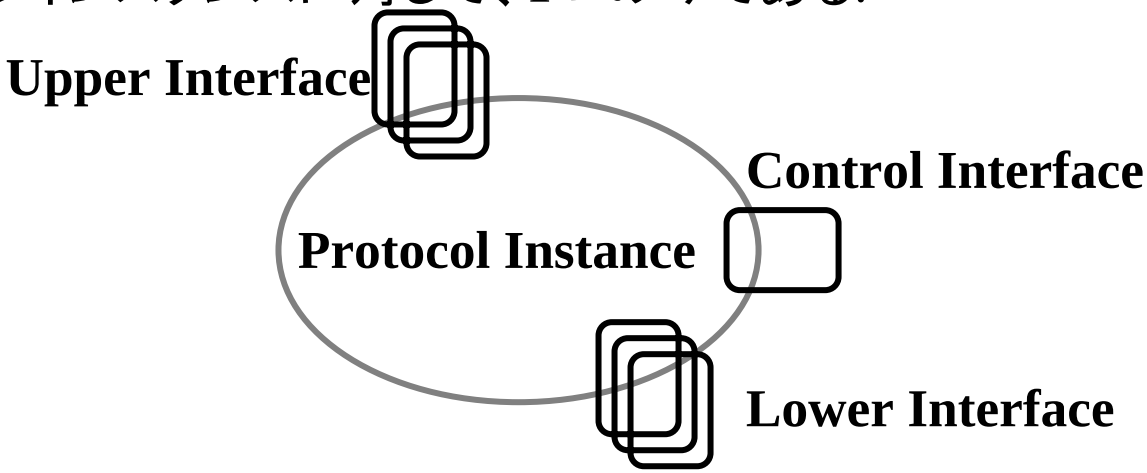
プロトコルインスタンスに対しては、設定(Config)、連結(Connect)、切断(Disconnect)、有効化(Activate)、無効化(Deactivate)のメソッドを与える.

- Config
  プロトコルインスタンスの設定をする.
- Connect
  プロトコルインスタンス間を接続する.
- Activate
  プロトコルインスタンスを有効化する.
- Deactivate
  プロトコルインスタンスを無効化する.
- Disconnect
  プロトコルインスタンス間を切断する.

プロトコルインスタンスのインターフェイス

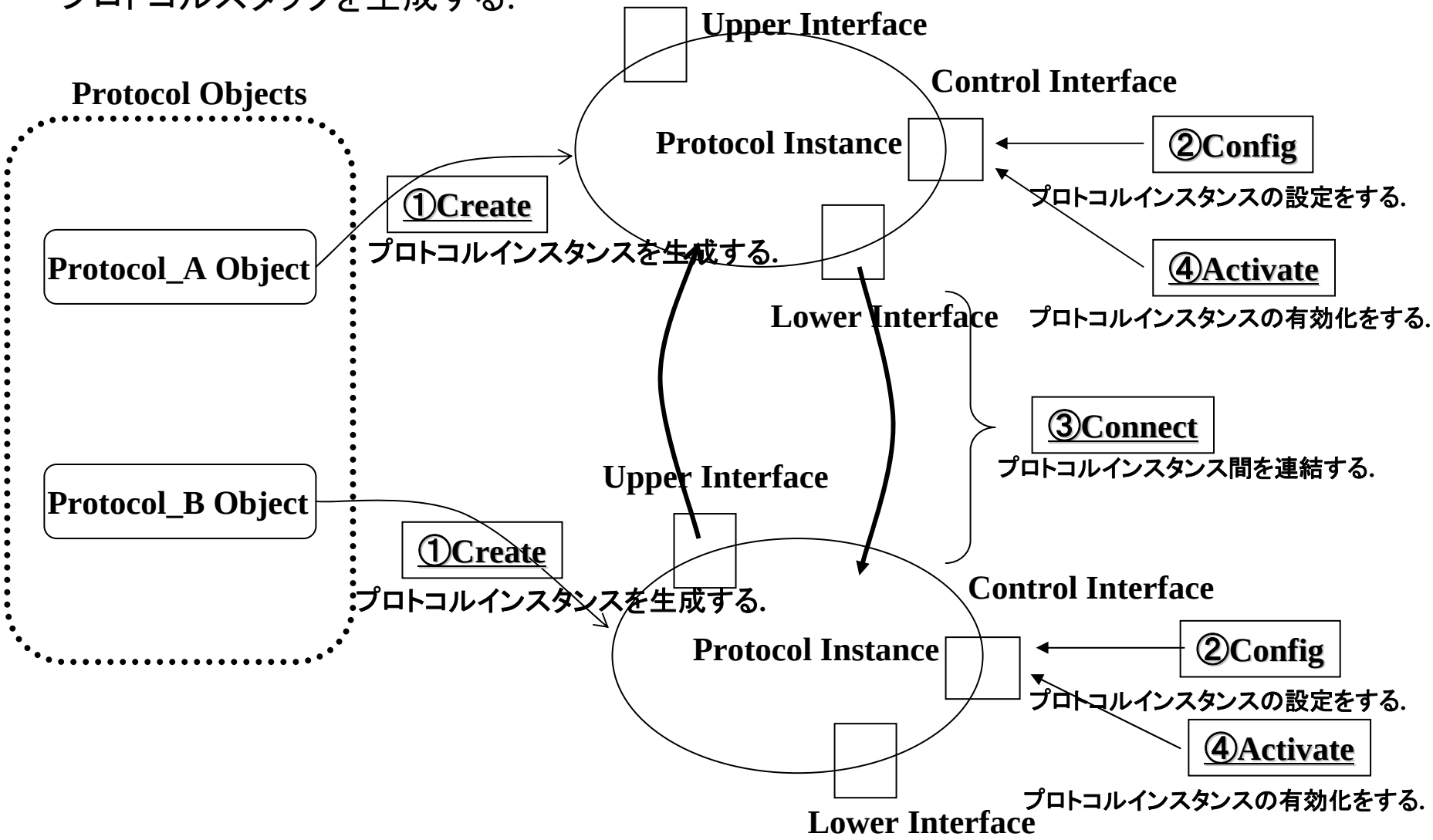
各プロトコルインスタンスは、以下の3種類のインターフェイスを備える.

- Upper Interface
  - 上位プロトコルインスタンスに対するインターフェイスである.
  - プロトコルオブジェクトの提供するサービスに応じて、複数になる場合がある.
- Lower Interface
  - 下位プロトコルインスタンスに対するインターフェイスである.
  - プロトコルオブジェクトの提供するサービスに応じて、複数になる場合がある.
- Control Interface
  - プロトコルインスタンス自体の制御インターフェイスである.
  - プロトコルインスタンスに対して、1つのみである.



プロトコルスタックの生成方法

図のようにCreate、Config、Connect、Activateを繰り返すことにより、
プロトコルスタックを生成する.



VII まとめ

David Cutler のソフトリアルタイム処理モデルを説明した。

V2.6 Linux カーネルも、このモデルにより、リアルタイム処理を組み込むことが可能。

多様なプラットフォームで、非同期処理をベースにした、大規模・高性能かつ柔軟なデザインパターンを、実装例とともに蓄積していくことが、重要。

付1：Windows NT ファミリーのコンテキスト

[カーネルモード・コンテキスト]

* デバイスコンテキスト： NSE，CPU固有コンテキスト

ハードウェア割り込みにより、非同期に起動されたコンテキスト。デバイス制御のうち、もっともタイムクリティカルな処理をISRにて行う。

ISR (Interrupt Service Routine)

現在のOSでは、デバイスコンテキストの処理をできるだけ縮小する方向にある。これは、優先度逆転が発生しやすいのと、過負荷時にシステムをロックする可能性があるため。

デバイスコンテキストでは、ページング可能なメモリ領域はアクセスできない。

* **DPC (Deferred Procedure Call) コンテキスト:**

NSE, CPU固有コンテキスト

すべてのハードウェア割り込みが消失して、カーネルで割り込み可能状態で実行されるコンテキスト。プロセススケジューラは動かず、プロセスコンテキストは割り込み時のまま。

DPCは優先度制御されるが、横取りなしでシリアライズされて実行される。

複数CPU時は、並列実行されうるので、spin ロックの保護が必要。

DPCコンテキストでは、ページング可能なメモリ領域はアクセスできない。

デバイスコンテキストから起動されたDPCは、デバイスコンテキストと同一CPUが割り当てられる。

- システム過負荷時の制御された縮退機能確保に大きな威力を発揮する。（Kernel Fork, Bottom-half, Tasklet, ソフト割り込み）

この場合、デバイスの割り込みが発生すると、その時点でデバイス割り込みを禁止し、あとの処理は、すべてDPCないし、カーネルスレッド（パッシブレベル）で実行する。

***APC: Asynchronous Procedure (Up)Call**　**SE,** 一般コンテキスト

- 入出力の終了やタイムアウトなどのイベント発生を、プロセスに非同期に割り込む手続き呼び出しでサービスプロバイダ側から通知する。出口処理、AST (Asynchronous System Trap)。Real-time Signal（後述）も同様なコンテキストと考えられる。
- Windows では、APC は種類が3種類に増え、ユーザモードAPCは、スレッドが alertable という特別な状態（事実上、イベント待ちといっしょ）の時のみに配送される。したがって、名前とは裏腹に非同期性はない。

***パッシブレベルコンテキスト:**　**SE,** 一般コンテキスト

- プロセススケジューラによりディスパッチされたカーネルモードプロセス。これは、さらに、特定プロセスのプロセスディスクリプタやリソースへのアクセスが必要な「プロセス指定コンテキスト」と、同一同期プリミティブで待っている任意のプロセスでかまわないとする「任意プロセスコンテキスト」とがある。
- ユーザプロセスに I/O 終了などを通知したい場合は、「プロセス指定コンテキスト」にし、指定されたプロセスが選択・実行されている状態にする。

典型的なシステムルーチンの分類例

Routine	Caller's Context (IRQL)	Thread context
AdapterControl	DISPATCH_LEVEL	Arbitrary
AddDevice	PASSIVE_LEVEL	System
BugCheckCallback	HIGH_LEVEL	Arbitrary: depends on state of operating system when the bug check occurred
Cancel	DISPATCH_LEVEL	Arbitrary
ControllerControl	DISPATCH_LEVEL	Arbitrary
DispatchCleanup	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchClose (for FSD, FS filters, and other highest-level drivers)	APC_LEVEL	Arbitrary
DispatchClose (for all other drivers)	PASSIVE_LEVEL	Arbitrary

典型的なシステムルーチンの分類例

Routine	Caller's Context (IRQL)	Thread context
DispatchCreate	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchQueryInformation	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchRead	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchSetInformation	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchShutdown	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers

典型的なシステムルーチンの分類例

Routine	Caller's Context (IRQL)	Thread context
DispatchSystemControl	PASSIVE_LEVEL	Arbitrary
DispatchWrite (for devices not in paging path)	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DllInitialize	PASSIVE_LEVEL	System or arbitrary
DllUnload	PASSIVE_LEVEL	Arbitrary
DpcForIsr	DISPATCH_LEVEL	Arbitrary
DriverEntry	PASSIVE_LEVEL	•System
InterruptService	DIRQL for the associated interrupt object	Arbitrary
IoCompletion	<= DISPATCH_LEVEL	Arbitrary
IoTimer	DISPATCH_LEVEL	Arbitrary
Reinitialize	PASSIVE_LEVEL	System

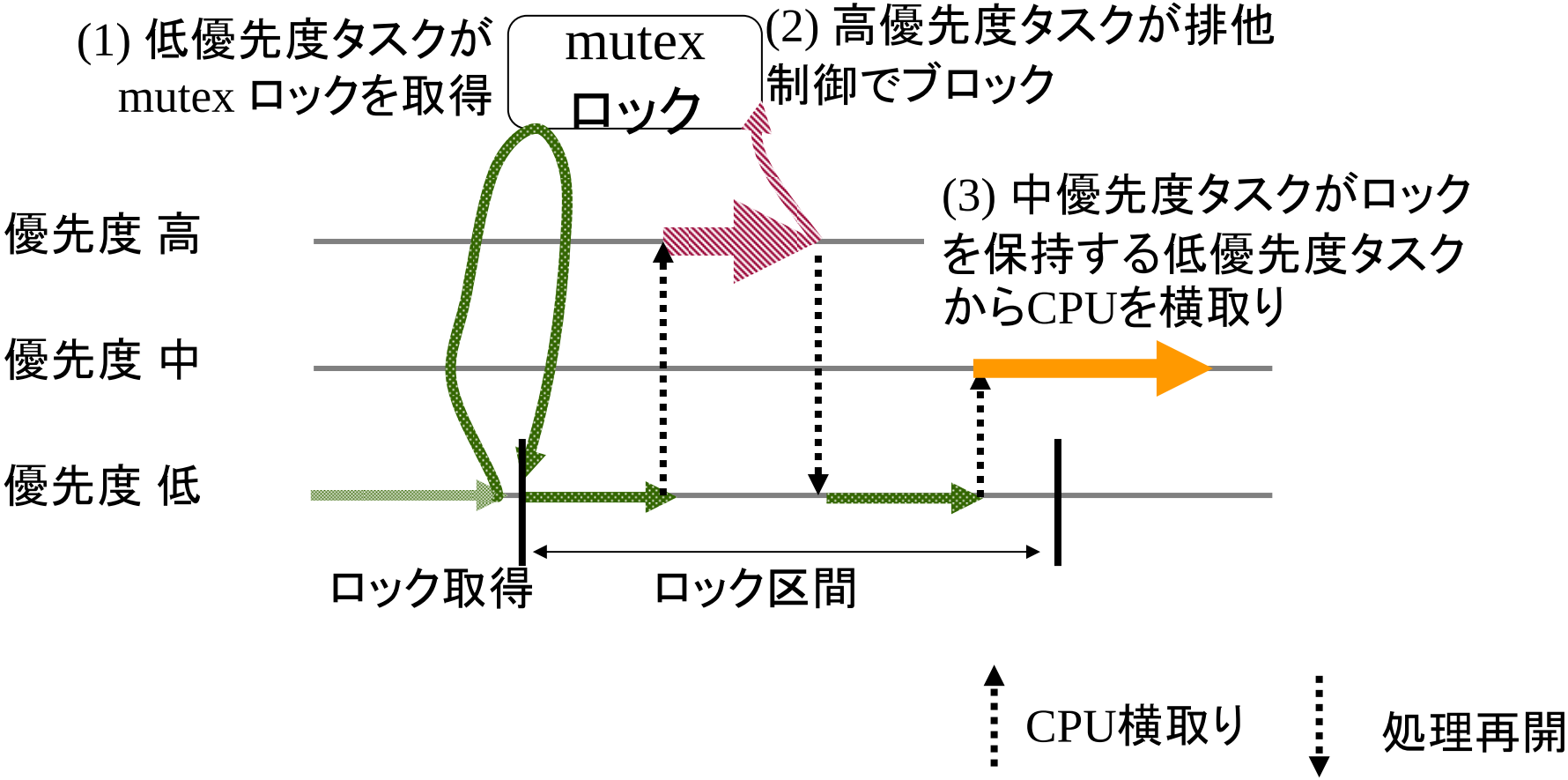
典型的なシステムルーチンの分類例

Routine	Caller's Context (IRQL)	Thread context
StartIo	DISPATCH_LEVEL	Arbitrary
SynchCritSection	DIRQL for the associated interrupt object	Arbitrary
Unload	PASSIVE_LEVEL	System

Scheduling, Thread Context, and IRQL,
Microsoft Windows Hardware and Device Central (2007)
(http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/IRQL_thread.doc)

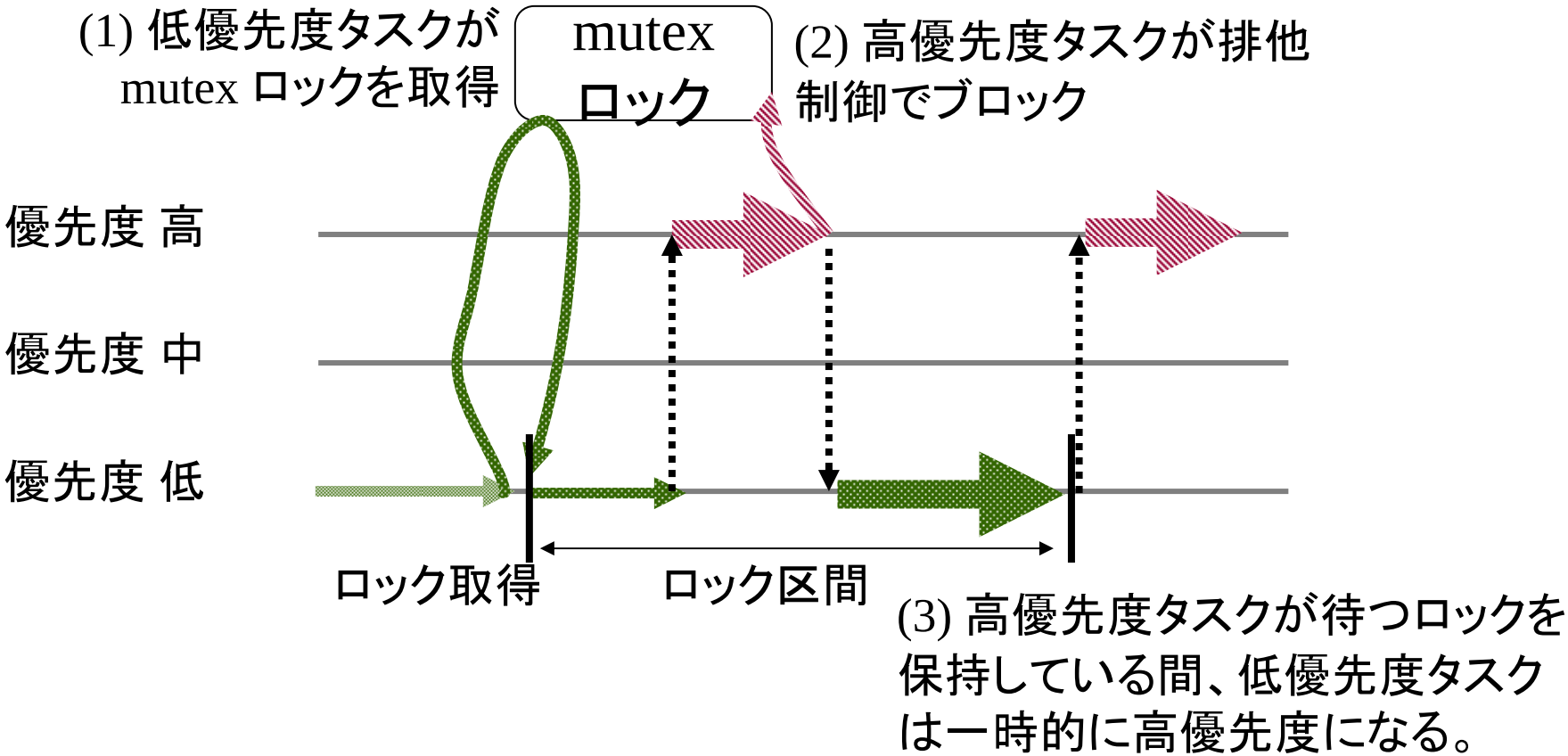
付2: Priority Inversion

下記の(1) - (3) の事態で、高優先度タスクは、ロックが解放されないため、永遠に動けなくなってしまう。



Priority Inheritance (優先度継承）

•ロックの保持者は、そのロックの解放を待っているタスクの最高優先度を一時的に継承する。



Priority Ceiling (優先度上限)

•ロックの保持者は、そのロックを使う可能性があるタスクの最高優先度を一時的に継承する。